Software to translate TELCOMP programs into KDF9 ALGOL

By K. A. Mulholland*

A translator has been written that enables programs written in TELCOMP and dumped onto paper tape to be translated into KDF9 ALGOL programs also on paper tape. The paper describes the methods adopted in dealing with the various syntax analysis and translation problems that were encountered in writing the translator.

(Received January 1969)

This paper describes a program written almost entirely in KDF9 ALGOL that will translate a source program written in one of the currently operational conversational languages TELCOMP into an object program in ALGOL text on paper tape ready for running. This allows the preparation of fault-free programs by means of conversational mode programming which may then be run on a larger, faster machine capable of accepting ALGOL.

The translator is a two pass translator. In the first pass the standard functions are rendered into their equivalent ALGOL form or, if no equivalent exists, a procedure body is output on paper tape for inclusion in the object program.

The various declarations **real** or **array** required by the object program are detected in the first pass and stored in a numerical form ready for output to the object program when the first pass is complete. To avoid overfilling the core store when a long program is to be translated each line of the source program is dealt with individually and then stored on magnetic tape.

Aims of the program

The primary object of the program was to accept any source program in TELCOMP 1 or 2 without restriction.

The second aim was that the resulting ALGOL object program should be as close to normal ALGOL usage as would be consistent with the primary aim.

The read procedure

The TELCOMP programs are run using a TELETYPE terminal, these terminals produce non-fixed parity ISO code 8 hole paper tape. The English Electric KDF9 ALGOL compilers do not provide facilities to read mixed parity paper tape, and consequently input procedures were written in 'USERCODE' which is a low level language for the KDF9 computer. The input TELETYPE character code is translated to the internal KDF9 ALGOL basic symbol code by a directly addressable look-up table, operating during the transfer process. The look-up table is accessed by using the arbitrary numeric value of the input characters as a modifier address acting on the base address of the look-up table. Any odd characters, line feeds, carriage returns and erases are removed. The TELETYPE 'Control S' character is used to identify the end of a TELCOMP line of program.

The TELETYPE paper tape character code has variations between the directly keyed program and that dumped up the line from the computer. Both codes have been made to produce a common sequence of basic symbols.

The read in process is terminated by the presence of the word 'DUMPED' after a 'Control S' character which is the standard TELCOMP program terminating instruction.

Types of instruction available in TELCOMP

Each line of a TELCOMP program has a label. This is called a step number and it can lie in the range 1 to 99·99999. This step number is followed by one instruction only. The instructions allowed are described in the TELCOMP manual (1967).

Most of these instructions can be followed by a list of expressions separated by commas which are controlled by the instruction. These are explained as follows:

SET

SET A=B, C=1 × SIN(TH), D=LN(1+E)

These are assignment statements that when written in ALGOL 60 would appear thus:

$$A:=B;\ C:=1+\sin(TH);\ D:=\ln(1+E);$$

DEMAND and READ

The operation of the instructions DEMAND and READ are similar. DEMAND interrogates the user for the value of a variable, READ reads the variable off paper tape.

The instruction:

DEMAND A, B, C

would appear in the object program as:

$$A:=read(20);\ B:=read(20);\ C:=read(20);$$

TYPE, PRINT, PLOT and SEND

The instructions TYPE and PRINT are output instructions that have a high degree of flexibility via a FORM (or Format) statement that controls the layout of a line of information output via the TYPE or PRINT instruction. PLOT is similar although the output is in graphical form. And SEND is a tape punching instruction that can punch any binary number in the range 0–255 on paper tape. These instructions can be matched by suitable ALGOL write commands except for the SEND facility which needs special treatment.

* *Department of Building Science, University of Liverpool*

DO and TO

The instructions DO and TO are jump instructions. As previously stated a TELCOMP program is divided into steps. A group of step numbers each having the same whole number part is called a PART. A DO instruction can call either a whole PART or a part of a PART or a single STEP.

The TO instruction is simpler in that control is not returned to the point at which the TO instruction occurred. Thus a TO statement can have only one expression following it and it is analogous to a goto statement in ALGOL 60.

DONE and STOP

DONE is an instruction that deems the current part number to have been done. Control is thus returned to the point at which the last DO statement occurred.

The instruction STOP stops the action of the program entirely irrespective of the nesting of DO statements that may be current.

Any of these instructions can be controlled by as many FOR or IF clauses as can be placed on one line. (Except that FOR cannot modify a TO instruction.)

For example, the TELCOMP instruction

TYPE A[I] IF A[I]> 1 FOR I=1:1:10, 20

would appear as in KDF9 ALGOL as

```
for I := 1 step 1 until 10, 20 do
  begin
  if A01[I] > 1 then
    begin
    write (device number, format statement, A01[I])
    end
  end
```

Standard functions

The standard functions are detected by searching along each line for the particular group of basic symbols that is required. For example detecting the group SIN *without* adjacent alphanumeric symbols would indicate a sine and the group would be replaced by sin. The group SINFUL would not cause any action because the N was immediately followed by the alphanumeric character 'F'. It is not advisable to translate all letters directly into lower case letters because of the existence of certain ALGOL reserved identifiers such as 'arctan' 'sign' etc. If one of these were used in the source program their presence in the declaration at the outer block level would preclude their use within the procedure blocks (also declared at outer block level) used for including the TELCOMP facilities, ATN, SGN, etc.

Forms

Forms are included in the object program at this stage by declaring them as procedure bodies with the relevant number of formal parameters.

Output of declarations

Before commencing the second pass the necessary declarations are made. This is done by decoding the numbers that represent the identifiers used. For one dimensional arrays and real quantities these names can be declared straight forwardly but for higher dimensional arrays such a straightforward declaration would call for an excessive amount of core store. However, it is not necessary to have a large area of store available because in the source program, although any combination of subscripts can be called whether by name or by value, it is very unlikely that more than a couple of hundred such combinations will be called for any given array. Therefore source program higher dimensional arrays can be declared as procedures instead.

For each two-dimensional array two procedures will be necessary:

(1) The first with two parameters is a **real procedure**:

SET A = A[A, B]	Source program
A := *A*02(*a*, *b*)	Object program

(2) The second procedure has three parameters and handles assignments to array positions:

SET A[1, 1] = A[1]	Source program
DEMAND B[1, 1]	
*A*05(1, 1, *A*01[1])	Object program
*B*05(1, 1, *read*(20))	

Similar procedures are used for the three and four dimensional arrays.

Both these procedures act on common areas of core store, each assignment produces an 'address' and a value. The address is a unique combination of the array position called and is used to denote that the corresponding value is associated with a particular higher array position. When a value is called, the real procedure searches for the corresponding address and is assigned the relevant value.

Second pass

In the second pass the lines of the program are brought down into core store, processed and output to the object program.

Each line is given a label '*L*ndd' where 'ndd' is a consecutive set of integers independent of the numbering of rows in the source program. When a change in part number is detected in a row of the source program an extra row is output to the object program:

goto *end*;

The whole object program is in the form of a procedure: **procedure** *part*(*x*); (or *part* (*x*, *bool*);) **real** *x*; **Boolean** *bool*; the **Boolean** *bool* is included if a statement DO STEP is found in the source program and in this case each line is terminated by the statement

if *bool* **then goto** *end*;

When a statement DO PART N is used in the source program this is written as:

part(*N*, **false**);

in the object program. The procedure is thus re-entered. The first line of the program is a statement:

```
switch L := L1, L2, . . . , Lndd;
enter: for i := 1 step 1 until LabN do
  begin if x ⩽ label [i] and x > label [i − 1] then
    goto L [i]
```

The switch is capable of sending control to any line of the object program within the current call. A TO statement is translated as

TO PART M
$x := M$; **goto** *enter*;

The DO STEP statement is identical with the DO PART except that the Boolean value is put **true**;

DO STEP 2	Source program
part(2, **true**);	Object program

The object program will look like this:

begin *open* (20); *open* (30); $L[0] = ddd$; $L[1] = 1{\cdot}01$;
$L[2] = 1{\cdot}02$; $L[ddd] := 9{\cdot}1$:

begin procedure *part* (*x*); **real** *x*;
begin switch $L := L001, L002, \ldots, Lddd$;

enter: **for** . . .
*L*001:
*L*002:
*L*003:
goto *end*;
*L*004:
*L*005:
goto *end*;
Lddd:
end:

end; *part* (1) **end**; *stop*: *close* (20); *close* (30)
end→

If a DO STEP statement occurs in the source program the modifications indicated above are included.

An individual line is output in the following manner:

A comment introduced by a semicolon that is not included in string quotes is output first and the line length reduced to eliminate the comment from the source program.

Output of IF and FOR statements

The line is then scanned for IF and FOR statements. The symbols are scanned in reverse order until the group of symbols IF or FOR is detected (once again care is taken to avoid scanning within string quotes).

The IF statement can be output

IF . . .
if . . . **then begin** . . . **end**;

The Boolean expression following the if statement cannot be directly output symbol by symbol because some of the relational operations have to be changed:

Source		*Object*
:=		=
>:=	:=>	⩾
<:=	:=<	⩽
< >	><	≠

Also the delimeters AND, OR and NOT must be searched for and changed to **and or not.** This is all achieved by straightforward syntax analysis using a number of if statements controlled by a for statement. The fact that the final **end** will be needed is noted by increasing the value of an integer *nend* by one from its initial value of zero. The FOR statement can be dealt with similarly:

SOURCE PROGRAM
FOR 1=1:1:100,200,300
OBJECT PROGRAM
for *i* := 1 **step** 1 **until** 100,200,300 **do begin** . . . **end**;
i := 300;

There is no difficulty here, the difficulty arises with the multiple increment FOR statement

FOR I=1:1:10:2:20

this has no direct equivalent in ALGOL but can be reduced indirectly thus:

for *I* := 1 **step** 1 **until** 10, 10 + (2) **step** 2 **until** 20 **do**

It is seen that to output this to the object program it is necessary to output two of the elements in the TELCOMP FOR statement twice. The FOR statement elements could be compound arithmetic statements of unknown length. The step by step output of for statement bodies is thus carried out as usual but as an arithmetic element is output each of its basic symbols is stored as successive terms in an integer array. If a third colon is found a comma is output followed by the third element which is reoutput by reference to the storage array. The symbols + (followed by the next FOR statement element followed by the symbol) **step.**

(The brackets are necessary in case the relevant element is signed negative), the fourth element is then output followed by 'until' in place of the TELCOMP ':'.

This process can be repeated indefinitely.

Besides IF and FOR clauses there is another optional clause controlling the action of the PLOT instruction.

For example:

PLOT A,B,1,0,1 ON F

values of F for the particular line of plotting are output in the left-hand margin of the display. Thus if the ON clause is found in a PLOT instruction a write instruction must be output to the object program.

DONE and STOP

The action of a DONE statement is to terminate the execution of the current PART of the source program. In the object program this is matched by transferring control to the end of the current call of the procedure 'part'. The instruction DONE is thus translated as **goto** *end*; the instruction STOP stops execution of the program absolutely. In the object program this is achieved by transferring control beyond the end of the procedure.

DO

If the first word is DO then the following translation is necessary:

DO PART 1, PART 2
part (1); *part* (2);

This recursive call of the program is adequate provided the instruction DO . . STEP does not appear anywhere within the program. If it does then the translation is:

DO PART 1, STEP 2·0
part (1, **false**); *part* (2, **true**);

This has been explained before. The main difficulty here is recognising the commas that separate the successive PART or STEP clauses. After the words PART or STEP have been recognised the program outputs all the source program up to an unbracketed comma or the end of the line. The test for an unbracketed comma is similar to that used in the first pass.

TYPE and PRINT

There are three sorts of TYPE instruction, *vis*:

TYPE A,B,C
TYPE A,B,C IN FORM N
TYPE FORM M

and there are three corresponding PRINT instructions.

The second sort is recognised by scanning along the line and finding the symbol 'IN', the third sort by finding 'FORM'. Separate routines are required to deal with each of the three sorts of type instruction.

The first sort requires that any expression in the list A,B,C . . . be output. These can be

(1) Arithmetic expressions.
(2) ≠ indicating a newline.
(3) Text (contained in string quotes).

These can be distinguished by examining the printing symbol following successive commas (or the initial TYPE). If it is a ≠ the instruction '*newline*(30,1);' is output to the object program. If the symbol is a string quote the string is output as part of the instruction '*writetext*(30,);'. If neither of these symbols is found then the symbols up to the next unbracketed comma is output in part of a write instruction

write (30,*f*, . . .);

When the instruction is TYPE a carriage return '*newline*(30,1);' instruction is output at the end of the TYPE sequence. If the instruction is PRINT no such instruction is output.

DEMAND and READ

Here the list of symbols are output individually each followed by the statement ':=*read*(20);'. The exception is the higher dimensional array. This is recognised by finding an '(' symbol unnested in square brackets the instruction output is then:

*A*05(*a*,*b*,*read*(20)); for a two-dimensional array.

TO statements

When a TO statement is found it is translated by a goto statement. This avoids unnecessary nesting of procedure calls that could lead to a run time failure if carried to excess.

TO PART N	Source program
x:=*N*; **goto** *enter*;	Object program

Ultimate control routine

Here the rest of the row is output making sure that unbracketed commas are translated into semicolons. Any necessary **end**'s are added and the instruction '**if** *bool* **then goto** *end*;' is added if it is needed.

The next line of source program is then read in. If the work 'DUMPED' is found the action of the translator is terminated by outputting the final statement. These are:

end:**end**;*part* (1); *stop*:
close(20); *close*(30)
end→

Here the statement part (1) is a call to the procedure that contains the program proper and is virtually the only program instruction not a procedure. It is assumed that programs are to be initiated by means of the instruction DO PART 1. If this is not so the 1 can be replaced by the relevant PART (or STEP) number. The labels *end* and *stop* are used in conjunction with the DONE and STOP statements.

Conclusions

Experience with running the program has been most successful. The trouble that has been encountered has been due to the attempt to implement the SEND facility. This has revealed inadequacies in the Kidsgrove ALGOL compiler which hitherto had been used at Liverpool when USERCODE subroutines were included in the object program which have so far proved difficult to overcome. The Kidsgrove compiler has been found to be unable to deal with statements of the form '**for** *L*:=1 **do**' and, of course, cannot deal with switches having more than 63 labels. The Whetstone compiler has none of these inadequacies but because of a local modification of doubtful value it cannot be used with USERCODE subroutines at Liverpool. However, it is hoped that shortly the EGDON program system will be implemented and the subroutines can then be kept permanently stored on disc and called by the main program compiled using the more flexible Whetstone compiler. Program translation time (at the moment about one line of TELCOMP source program per second) should also be improved.

References

(1963). Revised Report on the Algorithmic Language ALGOL 60, *The Computer Journal*, Vol. 5, pp. 349–367.
(1967). *TELCOMP 2 Manual*, Time Sharing Ltd., London.