A hardware representation for ALGOL 60 using Creed teleprinter equipment

By J. M. Gerard* and A. Sambles*

This paper describes an ALGOL 60 Hardware Representation for 5-hole paper tape Creed teleprinter equipment as used with Ferranti Pegasus and Mercury computers. This will be especially useful to those people expecting a KDF9 with Flexowriter who may wish to test ALGOL programs, since it has been designed to make the changeover as simple as possible. It also describes some of the problems of writing an Itemizer (as part of a Translator) to produce from the paper tape input the separate ALGOL items of a program (Identifiers, Logical Values, Delimiters, Numbers and Strings).

Introduction

The Hardware Representation which we shall describe was devised with several objects in mind. First, and most important, in spite of the extreme inadequacy of 5-hole paper tape, the teleprinted program should be as legible as possible. Secondly, it should agree with the KDF9 Flexowriter character set wherever this is possible. Thirdly, the number of letter shift and figure shift characters necessary should be minimized. We feel that the representation described below fulfils these requirements as well as can be expected.

The Hardware Representation

The actual details of this representation are as follows.

(1)	Underlined basic symbols. Precede with Asterisk, e.g. begin is *BEGIN		
(2)	Boolean Operators	$\neg$	*NOT
		$\wedge$	*AND
		$\vee$	*OR
		$\supset$	*IMP
		$\equiv$	*EQV
(3)	Semicolon	;	.,
(4)	Assignment	:=	.=
(5)	Base 10	$_{10}$	*v*
(6)	Exponentiation	$\uparrow$	*POW
(7)	Colon	:	$\rightarrow$
(8)	Integer Divide	$\div$	*DIV
(9)	Arithmetic Relations		
	As on the printer except:—		
		$<$	*$\geqslant$
		$\leqslant$	*$>$
(10)	Bracket Subscripting	[	*(
		]	)
(11)	String Quotes	`	£
		'	?
(12)	Delimiters already provided	.	$+ - \times / > \geqslant = \neq$, . ( )

* Now at University of London Computer Unit.

The reasons for the choices

(1),† (2) and (8) are as the KDF9 character set. Although some readers may feel that */ is preferable to *DIV for $\div$ this would only cause confusion in the change over to KDF9. In any case this symbol appears so infrequently that it is pointless to quibble over it.

(6) follows a suggestion by B. Higman (1962), and (11) follows an Elliott Brothers suggestion (Elliott Brothers, 1962).

(3) ., for ; . Although it may appear better to have a single character to represent semi-colon we found no suitable character, as all the available characters were illegible and/or easily capable of misinterpretation. The choice of ., was made because of its similarity to semi-colon and its effective separation of statements.

(4) .= for := . Here, again, the similarity between .= and := was a strong factor. The only single character worth considering was $\rightarrow$ and besides being very small on the teleprinter it is liable to misinterpretation in such statements as X $\rightarrow$ Y, which could be read as Y := X.

(5) *v* for $_{10}$. The obvious choices here were *r* and *n*. since we should strongly prefer a character on figure shift. The choice of *r* in preference to *n* was quite arbitrary. Though there is no apparent similarity between *r* and $_{10}$ we feel this will be immaterial in practice, since it will be used so often that it will rapidly become familiar, and it has no suggestive misinterpretation.

(7) $\rightarrow$ for : . This was chosen because the main use of colon is for labels and array declarations, and here $\rightarrow$ is found very suggestive in practice. The alternative of .. was rejected because we consider $\rightarrow$ better in view of its suggestive nature, and because .. has too much similarity to ., .

(9) *$>$ for $\leqslant$ and *$\geqslant$ for $<$. These were chosen in preference to such alphabetic representations as *LESS, *LESEQ, since the latter require letter shift and are clumsy and difficult to read. It may be noted from the specimen program below that their meaning is quite clear when a space is left at each side of them.

† We have since learnt that KDF 9 Flexowriter will have an underlining key for ALGOL basic symbols.

(10) *(for [and) for]. We chose these in preference to alphabetic representations for the same reasons as above. Although it may seem better to use *) for] we found that in practice this led to a meaningless jumble of characters. The asterisk is naturally read as attributable to the *pair* of brackets as a whole.

We have left *n* as an escape character.

To prevent possible errors we allow "." on either figure or letter shift to be the same, and leave to the Itemizer the task of discovering whether it is a full stop or a decimal point.

Spaces are allowed anywhere, for legibility.

A specimen program

To demonstrate what the teleprinted program looks like, we give the following ALGOL program, using KDF9 ALGOL Input and Output. This program illustrates all the points mentioned above.

```
*BEGIN *INTEGER N.,
  READ (N).,
                *COMMENT TO NORMALIZE A VECTOR.,
  *BEGIN *ARRAY V* (1 → N)., *INTEGER I., *REAL MAX.,
    MAX .= 0.,
    *FOR I .= 1 *STEP 1 *UNTIL N *DO
    *BEGIN READ (V*(I))..,
          *IF ABS(MAX) *⩾ ABS(V*(I)) *THEN MAX .= V*(I)
    *END MAXIMUM VECTOR ELEMENT IN MAX.,
    *IF MAX = 0 *THEN *GOTO FAIL.,
    *FOR I .= 1 *STEP 1 *UNTIL N *DO
    *BEGIN V*(I) .= V*(I)/MAX.,
          *IF V*(I) *⩾ v − 8 *AND V*(I) > −v −8
          *THEN V*(I) .= 0
    *END.,
    WRITE TEXT (£NORMALIZED VECTOR?)., WRITE CR(3).,
    *FOR I .= 1 *STEP 1 *UNTIL N *DO
    *BEGIN WRITE (£−D.DDDDDDv−DD? ,V*(I)) .,
          WRITE CR
    *END.,
    *GOTO OUT .,
  FAIL→WRITE TEXT (£ZERO VECTOR?)
  *END .,
OUT → *END
```

The Itemizer

In writing this we had two main aims. First, that any program which appears correct on the printed version of the input tape shall be interpreted correctly. This necessitates treating dot as the same on either figure or letter shift, and ignoring spare figure or letter shifts, etc. Secondly, the itemizer shall manage without having to store anything of the previous tape characters. This is possible if we have two stores, one for the last significant tape character (IT) and one for the next character on tape (ITN). By significant tape character we mean not one of LS, FS, LF, CR, Space (outside of strings), Erase. Connected with these we have two subroutine operations, "Next" and "Shift." "Next" puts the contents of ITN into IT and reads the next tape character into ITN. "Shift" puts the next tape character into ITN, but leaves IT unchanged.

The first difficulty encountered was what action to take when a "skip" or non-significant character may be present in ITN. This involves the idea of keeping count of lines for use in error printouts, and also of using a store SH for keeping a record of whether we are on letter or figure shift at present. The action is best explained in Fig. 1, a "Skip Action" flow diagram, to be entered when ITN may contain a skip character (except inside string quotes). It may be noted that the line counter "COUNT" is increased by one when we meet a sequence of skip characters containing at least one line feed, and a failure occurs when a CR but no LF occurs in the sequence.

Another difficulty we encountered was the elimination of comments after comment and end . We allowed an

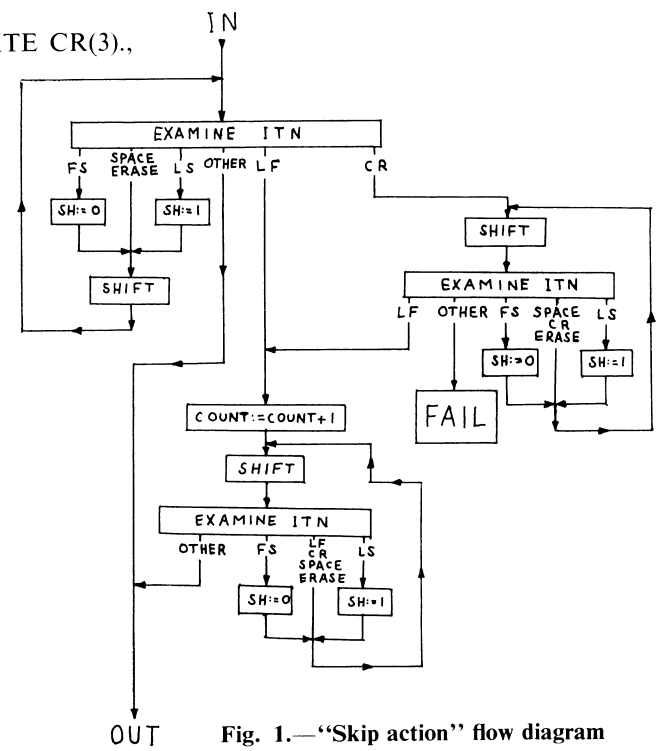

Fig. 1.—"Skip action" flow diagram

extension of ALGOL 60 by permitting comment to appear after any ALGOL item, instead of just after ; or begin as stated in the ALGOL 60 report [Naur, *et al.*, (1960)]. This was partly because it was easier than checking if the previous item to a comment was begin or ; . On meeting comment the itemizer puts out the first ALGOL item subsequent to the first semi-colon after this comment (unless this item is also comment).

We found further difficulty in the elimination of comments that may be written after end . When we had recognized a termination symbol (i.e. ., *END or *ELSE) we were left with a comma, D or E in ITN. Normally, on being entered from the translator the itemizer examines the next tape characters starting with the contents of ITN until it builds up an ALGOL item. In this case such a procedure would be wrong and we devised the following method to bypass this normal action, at the same time obtaining information as to which termination symbol is to be put out. This uses a machine word which is set negative in these circumstances, its magnitude indicating which termination character was used.

The identification of symbols preceded by asterisk is carried out by a dictionary method after preliminary elimination of *(, *$\geqslant$ and *$>$. That is, the first two significant characters after the * are compared with IF, DO and OR; if there is no check then the first three significant characters are compared with AND, END, FOR, etc. Finally the first nine significant characters are compared with PROCEDURE. If there is still no check a failure occurs. If, during this process, a significant character other than a letter is read in, a failure indication is given.

We chose the above method in preference to the alternative of letter-by-letter identification, since the latter is more involved and requires more storage space.

The bracket representation requires that we should have some special way of recognizing whether a closing bracket is actually a ")" or a "]". We considered two methods for this, both using a bracket counter BC.

In the first case, when "(" is encountered BC has one added; when "[" is met BC := BC $\times$ $2^5 + 1$. On meeting ")" we first test whether BC is zero and if so fail, since this bracket is then unmatched. If not we subtract one from BC and test the five least-significant digits to see if they are all zero. If so we put out "]" and do BC := BC/2^5, otherwise we put out ")". With a two-word store of 96 bits this allows 18 nested square brackets with 31 nested round brackets between any two consecutively nested square brackets. Failure occurs when this limit is violated. The method is both quick and relatively simple, allowing in the extreme case 18 $\times$ 31 nested brackets.

The alternative method uses BC as a push-down store, inserting a 1 for "[" and a 0 for "(". On meeting ")", the itemizer removes the available bit and examines it. If 1 then it puts out "]"; if 0 then it puts out ")". This is preceded by a check on surplus closing brackets using a second counter OBC, initially zero, which is increased by 1 at every opening bracket and decreased by 1 at every closing bracket. When this is examined as above and found to be zero a failure is given. This allows a nest of at least 48 brackets of any kind when BC is a single 48-bit machine word.

We feel that the choice between these two methods depends to a large extent on the facilities available on the particular computer used.

The fail action puts out the line number of the failure together with a failure number which indicates the type of error. On meeting a failure the itemizer ceases to send information to the translator, but continues to examine the program to detect any further errors. We feel that this action, although simple, gives sufficient information to the programmer to enable him to trace the errors.

Conclusion

We realize that at first sight the Hardware Representation may appear unsatisfactory in some respects. We would point out, however, that it was derived after a considerable amount of discussion and testing of ideas on a teleprinter. We feel that it is as good as possible with the available equipment. Unless some substantial improvement can be found we think it would benefit all concerned if this representation were to be generally adopted.

Acknowledgements

The work described here was carried out while the Authors were vacation students at the National Physical Laboratory, and this paper is published by permission of the Director of the Laboratory. The authors wish to thank Mr. M. Woodger for reading the manuscript and for his valuable assistance at all times while the above work was being carried out.

References

Elliott Brothers (London) Limited (1962). "The Elliott 803 ALGOL Programming System," list CS 134.
HIGMAN, B. (1962). "Towards an ALGOL Translator," *Annual Review of Automatic Programming*, Vol. 3.
NAUR, P., *et al.* (1960). "Report on the algorithmic language ALGOL 60," *Numerische Mathematik*, Vol. 2, p. 106.