An ALGOL line-syntax checker

E. W. Haddon* and L. G. Proll†

This paper describes a facility for partially checking the syntax of a line of an ALGOL program as it is input to a computer from a remote input device.

(Received March 1968, Revised July 1970)

This paper describes a line-syntax checker designed for use in a multi-access system. In such a system source code is usually stored in a source language file until the user requests an attempted compilation of the program. The line-syntax checker occupies a position between the user at his remote input device and the source file. Source code typed on the input device is transmitted into the system in groups of characters subject to some limit imposed by the computer hardware. Such a group of characters is referred to here as a line.

A line is passed to the line-syntax checker which determines, as far as is possible for a line *in vacuo*, whether the line is composed of valid source code. If it is, then the checker arranges for the transfer of the line to the user's source file. Otherwise, a descriptive error message is sent to the user and the faulted line is erased from the system. Although the syntax checker cannot find all errors in source code because it is working on a line *in vacuo*, we believe that it is a very useful feature of a multi-access system. The immediate verbal error messages with which the checker responds are more useful and instructive than the numeric error codes which are often output from compilers. A full compilation of an ALGOL program may, as a result of some errors, lead to the output of meaningless error messages as the compiler becomes confused. The checker, however, will give clear error messages about single lines and lead to the creation of a more correct source file which, in turn, will give a more useful first compilation. The time taken by the line-syntax checker to check one line is insignificant as far as the user is concerned. This checker was written for revised ALGOL-60 (1963); similar checkers could be written for other languages. Where necessary in this paper reference will be made to ICL 1900 computers and the ICL implementation of ALGOL.

General features of the checker and its design

The action of transmitting a message from an input device generates a request for some service by the system. It was found that this checker could process one line of input in about 0·5 millisecond. One copy of the checker should then be able to service the requests from many consoles without any apparent loss of response from the system. Since ALGOL is a completely format free language, the layout characters (space and newline) may appear anywhere within the significant characters of the language. Although most programmers, for readability, terminate a line after the statement terminator (;), it is not necessary for any line, with the exception of the last line of a program, to end with a completed statement. If checking is to be worthwhile it is essential that the checker knows whether the first character of a line is the first character of a statement or, if not, that the checker can continue checking at the relevant point. Thus the checker must preserve some information for each user who is currently availing himself of the checker's facilities. Upon getting a request for service, the checker must first ascertain whether the particular user last input a complete statement. If not, it must restore various parameters as they stood at the end of the previous line and resume checking at the appropriate point for continuation of the last statement. Ideally, one continuation store area should be available for holding continuation variables for each input device which is connected to the system, but this is not economic when storage space is at a premium. In practice the number of continuation store areas would be less than the number of connected input devices, the proportion being such that a user will not often be delayed because of the unavailability of a continuation store area. Users of the checker would soon realise that it was in their own interests to input complete statements whenever possible.

In order to keep the size of these continuation store areas as small as possible some restrictions have been imposed on the presentation of ALGOL to the checker. These are that all:

1. ALGOL basic symbols, **begin, real,** etc., and similar ICL program description symbols **list, program,** etc.
2. Identifiers.
3. Number structures.
4. Parameter delimiters, i.e. the comment separator of procedure parameters.

must be completely contained within one line of input. It is not expected that these restrictions will seriously inconvenience the user: a survey of users of normal batch input on cards has

Table 1 The variables whose values must be retained to check continuation lines

VARIABLE	FUNCTION
1	Indicates whether the line currently being checked is part of a comment.
2	Indicates whether the line currently being checked was preceded by a label or a 'FOR' statement or 'ELSE'.
3	Indicates the point in phase 3 at which to start checking a continuation line.
4-10	Used in searching the various transition tables and for holding return links from the routines for checking designational, arithmetic and Boolean expressions.
11	Holds a count of the numbers of string quotes encountered in a statement.
12	Holds the bracket and parenthesis count.
13	Holds a count of 'THEN' 'ELSE' pairs in conditional, arithmetic and Boolean expressions.
14	Similar to 13 for 'IF' 'THEN' pairs.
15	Similar to 13 for 'IF' 'THEN' and 'THEN' 'ELSE' pairs in conditional designational expressions.

*Formerly University of Southampton, now Computing Centre, University of East Anglia, Norwich
†Department of Mathematics, University of Southampton, Southampton

shown that (1), (2) and (3) is the normal practice and that (4) is a very infrequently used feature of ALGOL. The removal of these restrictions would lead to the continuation store areas doubling in size. Fifteen quantities are currently preserved in each continuation store area. The functions of these quantities are indicated in **Table 1.**

A further restriction, relative to the ICL 1900 ALGOL compilers, concerns those basic symbols for which there is an alternative representation. For example, 'LT' is a valid alternative for <, '**' is a valid alternative for ↑, etc. With the exception of ← (which exists on most teletype keyboards) as an alternative for the two-character symbol :=, only the more natural (ALGOL report) representations are recognised by the checker. Thus 'LT' would be faulted by the checker despite being acceptable to the compiler. These alternatives could easily be incorporated into the checker but have been omitted, partly for reasons of size and also because the authors believe that beginners should learn, and use, pure ALGOL.

Design of the checker

The aim of the checker is to implement as much of the ALGOL report as is possible when ALGOL code is being treated as statements *in vacuo*. Errors such as multiple use of identifiers or misplaced declarations cannot be detected by a line-syntax checker without the use of tables, etc., as in a compiler. This is impossible on two counts:

1. Much extra storage space would be required, both for users actively inputting ALGOL code and for incomplete dormant files.
2. The input of code for concatenation with other existing files or for replacement of sections of edited files would not have the necessary tables.

The structure of the checker is in three distinct, successive phases which conveniently allows for the possibility of overlaying to conserve core store.

Phase 1 checks whether a line is a continuation line, replacing the continuation variables if necessary or initialising the continuation variables in the checker otherwise. Its basic function is to scan the input buffer which contains the line, seeking space characters. Since these have no significance they may conveniently be removed, and the non-space characters are then compacted into the same input buffer. The number of non-space characters is counted for use in phase 2. If the line is not faulted during phases 2 or 3, only this compact, non-space line is transmitted to the file store. Storage space is thus saved at the expense of a compressed listing of the user's file, should one be requested. If any error is detected in phases 2 or 3, the error message refers to the compacted characters. A further advantage gained by this removal of space characters is that the subsequent phases do not have to contain code to detect and step past the space characters.

Phase 2 performs two functions simultaneously on a pass through the compact line produced by phase 1. The major part of this phase is a classification of the elements of the line into groups, where the elements of each group have the same syntactical properties as far as the checking performed by phase 3 can distinguish. The various groups are shown in **Table 2**. A single numeric character is set in a transformed line to indicate the type of element detected in the compact line. The basic symbols **begin, for** etc., are represented in 1900 ALGOL by the sequences 'BEGIN', 'FOR', etc. Phase 2 also checks these basic symbols for correct letter sequences: the opening apostrophe causes entry to a routine which calculates a unique numeric value for each symbol from the internal numeric values of the constituent letters. Exit from this routine occurs on detection of a closing apostrophe or on the elapse of 10 characters from the opening apostrophe (since the largest basic symbol is **procedure** having only nine letters). The

Table 2 The classification of elements of the compacted line

ELEMENT OF COMPACTED LINE	CHARACTER IN TRANSFORMED LINE
Identifier	0
Number structure	1
) {*if not followed by:*}	2
[	3
]	4
< > = ≠ 'GE' 'LE'	5
* / ↑ '/' {*integer division*}	6
+ −	7
'AND' 'OR' 'IMPL' 'EQUIV'	8
(	9
:	10
;	11
: = {*two characters*}←	12
'TRUE' 'FALSE'	13
'NOT'	14
' {*apostrophe, printed herein*} '*or*'	23
,) : comment (	28

numeric value so calculated is then sought in a table of the values of valid basic symbols (see **Table 3**). If the sequence between the apostrophes represents a valid basic symbol of the ALGOL report, a character to indicate a basic symbol and the position of the basic symbol within the table are deposited in the transformed line. ICL symbol and character representations which are also enclosed in apostrophes are checked by the same routine but lead to deposition of characters in the transformed line as shown in Table 2, with the exception of string quotes represented in 1900 ALGOL by '(' and ')'. On detecting a string the checker checks the whole string and if valid, records it in the transformed line as an identifier, since a string must be used as a parameter of a procedure. If no coincidence of the numeric value created from a sequence in apostrophes and the values in the table is found, the sequence is not valid. A mistake easily made by newcomers to programming and indeed, not infrequently found in programs prepared by more experienced programmers, is the omission of one of the apostrophes enclosing a basic symbol. This, in common with a number of other errors, can cause great confusion in an attempted compilation of a complete program with the possible masking of other errors in the program. The use of the line syntax checker will prevent such errors being presented to a compiler and therefore result in the elimination of many frustrating and useless compiler diagnostics.

When an error is detected in a line, either during phase 2 or phase 3, a descriptive error message is returned to the user. This will immediately follow the line in error and will quote four characters from the compact line to indicate the approximate position of the error. Since phase 3 works on the transformed line produced by phase 2, the latter must establish a mapping vector which gives the correspondence between the compact line and the transformed line. For each character in the transformed line there is an entry in the mapping vector giving the position in the compact line of the first character of the corresponding element (see **Fig. 1**). Phase 3, in checking the transformed line would detect an error at the 10th character of the transformed line. The mapping vector, position 10, contains 24 showing that the error occurs near the 24th character of the compact line. The user would then receive the message

ERROR IN ARRAY BOUNDS NEAR 4,CO

Note that one character preceding the position of the detected error is output in the error message. This is done because

Table 3 The basic symbols

	BASIC SYMBOLS	REMARKS		
1	REAL	Can be combined in various declarations		Allowable opening symbols of a statement
2	INTEGER			
3	ARRAY			
4	PROCEDURE			
5	BOOLEAN			
6	OWN			
7	COMMENT	Individual checking required		
8	BEGIN			
9	END			
10	FOR			
11	GOTO			
12	IF			
13	SWITCH			
14	EXTERNAL	Can only be followed by ;		
15	ALGOL	(ICL 1900-ALGOL symbols)		
16	STRING	Can only be followed by lists of identifiers		
17	LABEL			
18	VALUE			
19	STEP	Individual checking required		
20	UNTIL			
21	WHILE			
22	DO			
23	THEN			
24	ELSE			
25	GE		Classified in phase 2 (see Table 2)	Sequences of characters which are enclosed in apostrophes in 1900 ALGOL
26	LE			
27	TRUE			
28	FALSE			
29	AND			
30	OR			
31	IMPL			
32	EQUIV			
33	NOT			
34	/	'/' is integer division		
35	(	'(' and ')' are string quotes		
36	)			
37→	Program description symbols; treated separately			

Character position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
Line as typed		'	A	R	R	A	Y	'		M	A	T	R	I	X	[	1	:	R	O	W	,	−	1	4	,	C	O	L	]	;		
Phase 1:																																	
Compact line		'	A	R	R	A	Y	'	M	A	T	R	I	X	[	1	:	R	O	W	,	−	1	4	,	C	O	L	]	;			
Phase 2:																																	
Transformed line	23	3	0	4	1	10	0	28	1	28	0	3	11																				
Mapping vector	1	2	8	14	15	16	17	20	21	24	25	28	29																				

Fig. 1. Example of stages created in checking a line

several errors only become apparent when the next element is inspected.

The major syntactic checking is performed upon the transformed line by phase 3. If the line being checked is a continuation line, the continuation variables will have been restored and one of these, by the use of a multi-way switch, causes entry to phase 3 at the point reached by the previous line. The state of the continuation store area must be retained until the continuation line has been checked since if the continuation line contains an error the continuation variables must be retained for another attempt at a correct continuatio n line. It is possible, however, for the mistake to lie, not in the continuation line but in an already accepted line: for example, the line

$$X := A + B$$

would be accepted and stored with continuation variables set for a continuation line. If the continuation line is then typed as

$$-D)*C;$$

an error is detected on examining the right parenthesis. If a left parenthesis should have preceded A in the first line the user must have a facility for direct input of his continuation line without checking. Subsequent editing, using normal facilities, could insert the left parenthesis.

If a line of input is not to be interpreted as a continuation line it must start a new statement and the only legal first characters of a statement are the apostrophe of a basic symbol or a letter. With reference to Table 3, only the first 18 basic symbols may open a statement. Thus if V is the position of a basic symbol within the list in Table 3, a value of $V > 18$ is illegal at the beginning of a statement. If the starting symbol is legal and $V > 15$ the checker proceeds to look for a list of identifiers. If not, but $V > 6$, control passes to individual checking routines for the appropriate structures. Any other value of V indicates a symbol which may be used singly or in certain combinations in type declarations or specifications, such as 'REAL' or 'OWN' 'BOOLEAN' 'ARRAY'. $V \leqslant 6$ causes entry to that part of the checker which checks the use of these symbols. This is achieved through the use of a transition table which is typical of the checking employed in various situations where elements may be connected in alternative sequences. The particular transition table and its method of use is shown in detail in **Figs. 2 and 3.** Entries in the transition table are selected according to the position reached via previous elements and the value of the next element of the transformed line. This entry is then tested to determine the next stage in use of the transition table or a branch to a further structure check or the occurrence of an error. The entries in these transition tables are only small integers and may conveniently be packed into store to provide a more compact checker (Day, 1970).

The structure of ALGOL makes it impossible or impractical to use large transition tables, for many elements of such tables would then denote illegal states. Much of the checking of phase 3 is performed through transition tables with particular

N \ V	REAL 1	INTEGER 2	ARRAY 3	PROCEDURE 4	BOOLEAN 5	OWN 6
1	2	2	−1	−2	2	3
2	0	0	−1	−2	0	0
3	4	4	0	0	4	0
4	0	0	−1	0	0	0

Fig. 2. Transition table used in checking some type declarations

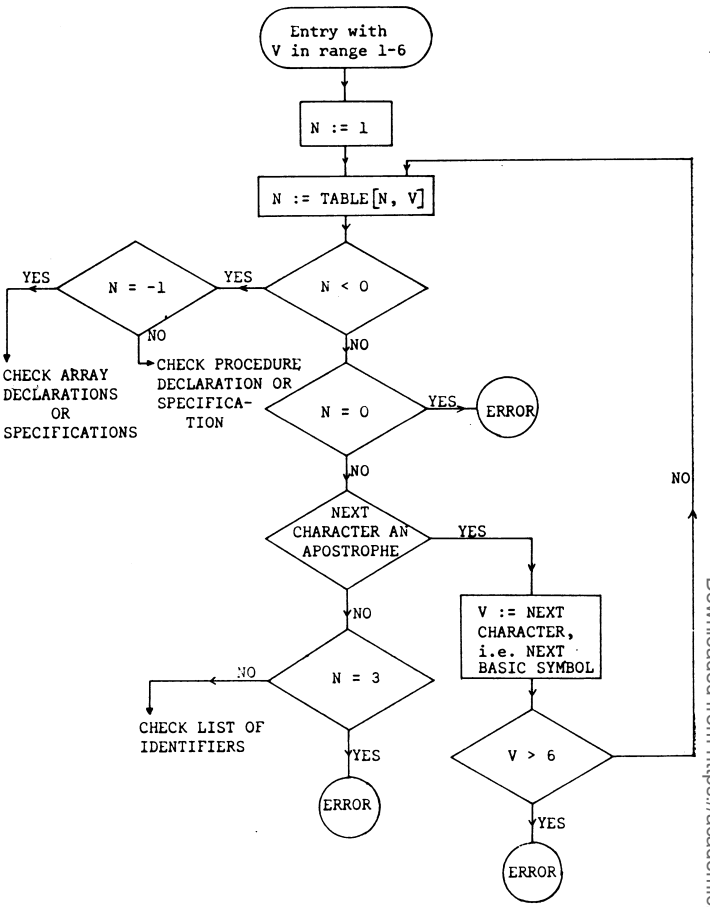

Fig. 3. Flow diagram illustrating the use of the transition table of Fig. 2

hand-coding being used to pass from one transition table to another. For example, 'ARRAY' must be followed by a list of identifiers. The identifier, however, may be terminated by a semi-colon if 'ARRAY' is being used in the specification part of a procedure declaration (and since the checker is working on lines, *in vacuo*, it does not know), or by the [of array bounds. This causes entry to a checking sequence for array bounds which requires access to checking of arithmetic expressions, including the possibility of nested conditional arithmetic expressions.

Switch declarations also show the same degree of complexity since although the elements of the switch list are designational expressions, which are generally simple labels (i.e. identifiers), the following structure

'SWITCH' S := S1, S2, Q[I*(J − 1) + P[3]],
'IF' Y > 0 'THEN' S3 'ELSE' S4;

is perfectly valid. Access is thus required to those sections of the checker designed to check arithmetic expressions, Boolean relations and the conditional 'IF' ... 'THEN' ... 'ELSE' structure. When the latter is used as a designational expression it is possible to check for the existence, and correct sequencing of the basic symbols 'IF' 'THEN' and 'ELSE', even if S3 and S4 are other than simple labels. The conditional statement is, however, more difficult to check in its general use since there are the two forms

'IF' R 'THEN' S1;

and

'IF' R 'THEN' S2 'ELSE' S3;

in which R represents a Boolean relation and S1, S2, S3 can take many forms. In these structures it is possible for the checker to determine the correct matching of 'IF' 'THEN' pairs. The checking of 'THEN' 'ELSE' pairs can only be taken as far as checking that the number of 'ELSE' symbols in

a statement does not exceed the number of 'THEN' symbols which have occurred. A further complexity of the second structure arises from S2 having the possibility of being a compound statement, e.g.

'IF' R 'THEN' 'BEGIN' A := B; B := 0 'END' 'ELSE' S3;

Since the compound statement may itself be extremely lengthy and of course, involve the same structure nested within itself, it is not practical for the checker to check the complete structure of a conditional statement. The decision was therefore made to check a conditional statement only as far as the first semi-colon. In this example therefore, checking re-commences with B := 0 as of a statement in isolation. The checker cannot thus determine whether 'ELSE' is being used correctly, though if 'ELSE' is detected, as here, with the 'IF' 'THEN' count at zero, the 'ELSE' must have been preceded by 'END' or by 'END' comment with no semi-colon between the 'END' and the 'ELSE'.

Perhaps the most difficult constituent of ALGOL as regards line-syntax checking is the checking of expressions. Complete checking of an expression when taken out of the context of the program is impossible—for instance, TEMP(X + 2) is valid if TEMP is a procedure identifier but not otherwise.

The basic philosophy of expression checking is that a transition table is used to determine whether one element of the expression is a legal successor of the previous element. Obviously, this will not trap all errors, for example

X := A + B 'OR' C;

would not be faulted. More thorough checking of expressions could have been introduced, even without detailed analysis of the expression, but only at the expense of much larger transition tables, increases in the size of the continuation store areas and extra hand-coding for intermediate checking within the expression. A consideration in favour of this reduced form of checking for expressions was the result of a survey of user programs being presented to the batch ALGOL compiler. This showed that the majority of errors in arithmetic expressions which could be detected in a line, *in vacuo*, would, in fact, be trapped by this simple method.

One relatively common error in expressions which the above checking technique can not detect is the imbalance of parentheses and brackets. This is of sufficient occurrence to warrant a special check and hence the line-syntax checker will determine the correct sequencing and matching of parentheses and brackets. For example, the statement

X := A + B(C, D(E[2, 3 + X)], F);

would be faulted. Only one variable is used to hold the count of parentheses and brackets for any conceivable depth of nesting.

The checker and its environment

The checker was designed to be one of a set of interacting modules forming a multi-access system. The major module of such a system would be a command interpreting module (CIM), to detect input commands from the attached remote devices and instigate action in other modules. A command verb SYNCHECK input from a remote device could be used to inform the CIM that until another command verb was input from that device each line of input was to be held in a buffer for servicing by the syntax checker. The checker would, when free, check the contents of the buffer and leave a flag set in one of two states to inform the CIM that the contents of the buffer, i.e. the compact line, were now to be added to that user's file and the user invited to input another line or that the line contained an error. In the latter case the checker would have set up some coded error message together with the four characters from the compact line. The CIM could now call in a message sending module which would assemble the verbal message, according to the code provided by the checker, from a system dictionary and transmit the error message to the user, followed again by an invitation to input another line.

The checker could alternatively be set up as a private program to assemble files of checked lines of ALGOL program in user's files which could then be processed in an existing system such as MINIMOP.

The checker has been coded in 1900 PLAN and, without overlaying, occupies about 1·5 K words. With overlays and a more careful coding the checker should require only about 1 K words of store.

Conclusions

We believe that such a syntax checker can be a useful feature of a multi-access system. The newcomer to ALGOL programming would find the error messages both instructive and time-saving. It is also suggested that experienced programmers would use the facility since the use of the checker rather than a direct mode of input would not cause a noticeable increase in input time. Moreover it offers two distinct advantages over the direct mode of input: first, that any errors detected by the checker can be corrected immediately simply by re-typing a corrected line. If the errors were only detected during an attempted compilation they would have to be removed by use of the editing facilities which would involve extra work. Secondly, by using the checker, the user has an increased probability of a successful compilation at the first attempt and, even if his program still contains errors, the first compilation might then detect more errors than would otherwise be possible.

These advantages to the user are also helpful to the system, the first since editing requires both time and extra storage space for edited files and secondly there will be fewer compilations.

As has been stated previously, the checker cannot detect all the errors in the statements presented to it. It is asserted, however, that no correct ALGOL line will be faulted (though some ICL departures from the ALGOL report such as 'FOR' I := 1 'STEP' 1 'WHILE' B 'DO' S; have purposely not been implemented). The checker has been thoroughly tested as a free standing program using a sample of users' programs (including experiences users) submitted for the batch compiler, as specimen test data; each card being interpreted as one line of input. The results obtained from the checker were compared with the output from attempted compilations. These tests showed that the checker was finding 76% of the independent errors in the programs, the other 24% of the errors being mainly due to errors in declarations and block structures which the checker could not detect by its very nature. The ALGOL compiler was, however, only finding 61% of the independent errors—a worse performance due basically to the masking of some errors in the confusion caused in the compiler by previous errors in the programs. The use of the checker followed by an attempted compilation should result in a significantly higher proportion of independent errors being extracted by the end of the first compilation.

Acknowledgements

The authors wish to thank the referee for his helpful comments and suggestions. We are also indebted to various colleagues with whom we have had valuable discussions.

References

(1963). Revised report on the algorithmic language ALGOL 60, *The Computer Journal*, Vol. 5, pp. 349-367.
(1965). ICL 1900 series ALGOL manual. Technical publication 3340.
DAY, A. C. (1970). The use of symbol-state tables, *The Computer Journal*, Vol. 13, pp. 332-339.