

Programming Languages

T. E. CHEATHAM, JR., Editor

An Error-Correcting Parse Algorithm

E. T. IRONS

Institute for Defense Analysis, Princeton, N. J.

During the past few years, research into so-called "Syntax Directed Compiler" and "Compiler Compiler" techniques [1, 2, 3, 4, 5, 6] has given hope that constructing computer programs for translating formal languages may not be as formidable a task as it once was. However, the glow of the researchers' glee has obscured to a certain extent some very perplexing problems in constructing practical translators for common programming languages. The automatic parsing algorithms indeed simplify compiler construction but contribute little to the production of "optimized" machine code, for example. An equally perplexing problem for many of these parsing algorithms has been what to do about syntactically incorrect object strings. It is common knowledge that most of the ALGOL or FORTRAN "programs" which a compiler sees are syntactically incorrect. All of the parsing algorithms detect the existence of such errors. Many have considerable difficulty pinpointing the location of the error, printing out diagnostic information, and recovering enough to move on to other correct parts of the object string. It is the author's opinion that those algorithms which do the best job of error recovery are those which are restricted to simpler forms of formal languages.

The algorithm presented here is the outgrowth of an attempt to alleviate some of these difficulties in error detection and recovery. Its general characteristics are:

(1) It will parse strings describable in essentially Backus Normal Form (BNF) [7, 8]. No automatic parse of the author's acquaintance will work for substantially more complicated languages.

(2) If an incorrect object string is presented to the algorithm, it will make local insertions, deletions or substitutions in the object string until a syntactically correct string is produced. Many errors made in such a way that the "correction" is clear from context will be corrected. In any event, no matter how garbled the object string is, it will be manipulated until a correct string has been obtained.

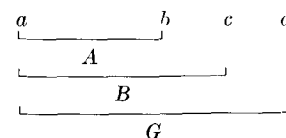
(3) The algorithm is relatively efficient. Pilot Models indicate that parsing proceeds at the rate of about 100 executed machine instructions per symbol of the object string.

(4) The algorithm is economical of memory space. In particular its intermediate storage requirements are quite restricted.

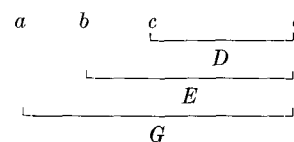
The essentially novel characteristic of the algorithm is that in parsing the object string (say from left to right) when a situation arises where more than one parse is possible for the next few symbols *all* possible parses are carried along until a symbol is reached which "selects" one of the parses. The following example will serve to illustrate this principle. The BNF grammar

$$\begin{aligned}\langle A \rangle &::= ab & \langle D \rangle &::= ce \\ \langle B \rangle &::= \langle A \rangle c & \langle E \rangle &::= b \langle D \rangle \\ \langle G \rangle &::= \langle B \rangle d & \langle G \rangle &::= a \langle E \rangle\end{aligned}$$

assigns the parse



to the string *abcd*, and the parse



to the string *abce*. This grammar presents a problem to a left-to-right parse because regardless of what string may occur to the left, the parse of *abc* cannot be determined until the next symbol after *c* is encountered.

There are essentially two ways in which this dilemma has been resolved.

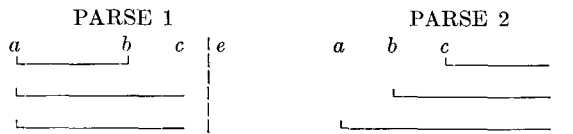
(1) The grammar is restricted so that a unique parse for a string *A* is determined by considering only the strings to the left of *A* and one symbol to the right.

(2) The parsing algorithm makes an assumption that one of the possible parses is correct, and if this turns out not to be the case, the algorithm back tracks and tries another parse.

The disadvantage of the first solution is simply that the parsable languages are from a considerably more restricted class than even BNF specified languages.

The disadvantage of the second solution is that in leaving the door open for back tracking, the occurrence of an error requires that a whole host of unexamined alternatives must be examined before it can definitely be established that an error has occurred. Furthermore when all alternatives *have* been so examined, the matter of deciding which unsatisfied alternative is unsatisfied *because of the error* is somewhat more than hopeless.

In the algorithm presented here, all possible parses are carried along as shown below in the progressing parse of *abce* according to the syntax of the earlier example.



When the symbol *e* is encountered, Parse 1 cannot be continued and is dropped, leaving Parse 2 as the correct one.

Because the parse proceeds in this way, the location of an error is easily detected, namely at the point where *no* parses can be continued. Error recovery is then effected by examining the next few symbols in the object string in relation to the syntactic statements concerning the parse "brackets" which have been extended up to the point of error. A more detailed discussion of the error recovery feature will be postponed until a more detailed description of the algorithm has been presented.

The Parse Algorithm

In order to describe the algorithm we present first the form of the metalanguage, used to specify the parsing and the way in which the statements of the metalanguage are stored in the machine.

We adopt as metasymbols those used in BNF, namely $\langle \rangle$ and $::=$, plus two braces $\{ \}$. The statements of the metalanguage take the form of BNF statements with the following restriction: No syntactic variable may occur both as the defined variable (left of the $::=$) and the first defining variable (immediately to the right of the $::=$ or $|$) nor may any set of statements exist such that a variable is defined in terms of itself. For example

$$\langle \text{TERM} \rangle ::= \langle \text{TERM} \rangle \langle \text{MULT OP} \rangle \langle \text{PRIMARY} \rangle.$$

is not allowed nor are the set of statements

$$\langle A \rangle ::= \langle B \rangle \langle C \rangle$$

$$\langle B \rangle ::= \langle A \rangle \langle D \rangle.$$

Having thus stripped BNF of all its recursive power by restriction 1, we add instead an "iterative" power by introducing the metasymbols $\{$ and $\}$ as follows:

Any set of syntactic variables embraced by the braces $\{ \}$ are specified to occur any number of times in an input

string. For example

$$\langle \text{SUM} \rangle ::= \langle \text{TERM} \rangle \{ \langle \text{MULT OP} \rangle \langle \text{TERM} \rangle \}$$

specifies that a $\langle \text{SUM} \rangle$ may consist of a $\langle \text{TERM} \rangle$ alone or a $\langle \text{TERM} \rangle$ followed by any number of occurrences of the pair $\langle \text{MULT OP} \rangle \langle \text{TERM} \rangle$. A final restriction prohibits a brace from occurring immediately after the $::=$ i.e.,

$$\langle A \rangle ::= \{ \langle B \rangle \} \langle C \rangle$$

is not allowed.

Without bogging down in comparisons of this metalanguage to BNF and others, we assert that as a practical metalanguage it is essentially as powerful as BNF and furthermore lends itself to somewhat more compact descriptions of languages. To reinforce this point we present the syntax in our metalanguage for a part of the arithmetic section of ALGOL 60 which we shall continue to use in later examples.

$$\begin{aligned} \langle \text{LETTER} \rangle &::= A \mid B \mid C \dots \\ \langle \text{DIGIT} \rangle &::= 0 \mid 1 \mid 2 \dots \end{aligned}$$

1. $\langle \text{IDEN} \rangle ::= \langle \text{LETTER} \rangle \{ \langle \text{LETTER} \rangle \} \{ \langle \text{DIGIT} \rangle \}$
 $\langle \text{ADOP} \rangle ::= + \mid -$
 $\langle \text{MULOP} \rangle ::= * \mid /$
2. $\langle \text{PRIMARY} \rangle ::= \langle \text{IDEN} \rangle \mid \langle \text{SUM} \rangle$
3. $\langle \text{FACTOR} \rangle ::= \langle \text{PRIMARY} \rangle \{ \langle \text{PRIMARY} \rangle \}$
4. $\langle \text{TERM} \rangle ::= \langle \text{FACTOR} \rangle \{ \langle \text{MULOP} \rangle \langle \text{FACTOR} \rangle \}$
5. $\langle \text{SUM} \rangle ::= \langle \text{TERM} \rangle \{ \langle \text{ADOP} \rangle \langle \text{TERM} \rangle \}$
6. $\langle \text{ADOP} \rangle \langle \text{TERM} \rangle$

The representation in the machine of these statements is designed to facilitate the parsing algorithm. In particular we wish to be able to assign the complete parse (or several of them) to a basic at the first moment it is encountered in the object string. To this end, construct from the syntax statements a "chain" table for each basic symbol as follows.

Observing that letter *A* can be the first symbol of a $\langle \text{LETTER} \rangle$, $\langle \text{IDEN} \rangle$, $\langle \text{PRIMARY} \rangle$, $\langle \text{FACTOR} \rangle$, etc. construct the chain

$$\begin{aligned} A \leftarrow \langle \text{LETTER} \rangle^0 \leftarrow \langle \text{IDEN} \rangle^{1.1} \leftarrow \langle \text{PRIMARY} \rangle^0 \\ \uparrow \\ \langle \text{SUM} \rangle^{5.1} \rightarrow \langle \text{TERM} \rangle^{4.1} \rightarrow \langle \text{FACTOR} \rangle^{3.1} \end{aligned}$$

for each letter. Five other symbols have chains:

$$\begin{aligned} + &\leftarrow \langle \text{ADOP} \rangle^0 \leftarrow \langle \text{SUM} \rangle^{6.1} \\ - &\leftarrow \langle \text{ADOP} \rangle^0 \leftarrow \langle \text{SUM} \rangle^{6.1} \\ * &\leftarrow \langle \text{MULOP} \rangle^0 \\ / &\leftarrow \langle \text{MULOP} \rangle^0 \\ (&\leftarrow \langle \text{PRIMARY} \rangle^{2.1} \leftarrow \langle \text{FACTOR} \rangle^{3.1} \leftarrow \langle \text{TERM} \rangle^{4.1} \leftarrow \langle \text{SUM} \rangle^{5.1} \end{aligned}$$

(Although for this example, it happens that each link of the chain has only one arrow pointing to it, there may, in general, be several arrows pointing to an element. There may be only one pointing away, however.) A chain for a symbol may be interpreted as indicating that the symbol may begin any syntactic category on its chain. Suppose, for example, we wish to know the parse of a $\langle \text{TERM} \rangle$ beginning with *A*. It is determined by looking for $\langle \text{TERM} \rangle$

on A 's chains, and following the arrows to A to construct

	A	
—	LETTER	0
—	IDEN	1.1
—	PRIMARY	0
—	FACTOR	3.1

The digits connected to the brackets (copied from the digits in the chain) are called "syntax pointers" and indicate elements of the syntax tree which effectively determine how the brackets may be extended to the right.

The syntax "tree" for our example would be

Index	Names	Alternates	Successors
0	null		
1.1	⟨LETTER⟩	1.2	1.1
1.2	⟨DIGIT⟩	0	1.1
2.1	⟨SUM⟩		2.2
2.2)		0
3.1	↑	0	3.2
3.2	⟨PRIMARY⟩		3.1
4.1	⟨MULOP⟩	0	4.2
4.2	⟨FACTOR⟩		4.1
5.1	⟨ADOP⟩	0	5.2
5.2	⟨TERM⟩		5.1
6.1	⟨TERM⟩		0

To interpret the tree, we adopt the following notation

S_i is the i th entry (line) of the tree table.

The *alternates* of S_i are $S_i, S_{i_1}, S_{i_2}, \dots, S_{i_n}$ where S_{i_1} is the alternate for S_i and $S_{i_{p+1}}$ is the alternate for S_{i_p} .

A bracket whose syntax pointer is i may be extended right one symbol if the next symbol has any of the alternates of S_i on its chain, and if all brackets "under" it can be terminated.

A bracket may be terminated if 0 (or null) is one of alternates of its pointer. Observe that for the parse

	A	
—	LETTER	0
—	IDEN	1.1
—	PRIMARY	0
—	FACTOR	3.1

⟨IDEN⟩ may be extended over a ⟨LETTER⟩ or ⟨DIGIT⟩ or since ⟨IDEN⟩ and ⟨PRIMARY⟩ may be terminated, ⟨FACTOR⟩ may be extended over ↑

Observe that if any bracket is extended,

- (1) all brackets "covering" it must be extended as well,
- (2) all brackets "under" it must be terminated,
- (3) the pointer for the extended bracket becomes the successor of S_i (where i was its old pointer),
- (4) if it is possible to extend two or more brackets, we must create a new parse for each extension.

Lest the workings of the algorithm be completely obscured by the above description, it is presented more precisely in the following (almost ALGOL) program.

We define the following arrays (with all lower subscript bounds = 1) and variables:

1. The chain for a symbol j

$CN[i, j]$ is the name of the i th element of the chain for the symbol whose numeric value (under some convenient mapping) is j .

$CS[i, j]$ is the *syntax link* (given as superscript digits in the earlier presentation) for the i th element of the chain for j .

$CP[i, j]$ is the index of the next element in the chain (and = 0 if the element is the last, namely the symbol j).

$NC[j]$ is the number of elements on the chain for j .

2. The syntax tree

$SN[k]$ is the name of the k th element of the tree table.

$SS[k]$ is the successor for this element.

$SA[k]$ is the immediate alternate, (if there is no alternate $SA[k] = 0$) $SN[1]$ is the "null" element.

3. The Parses

N is the number of parses currently existing.

$NP[n]$ is the number of brackets in the n th parse.

$PN[i, j]$ is the *name* of the j th bracket of the i th parse.

$PS[i, j]$ is the *syntax pointer* for the bracket.

$PI[i, j]$ is the index of the first (left most) symbol under the bracket.

Observing that once a bracket has been terminated we no longer need to keep it in the parse table, we may assign the following structure to PN (and corresponding parse vectors): For i th parse, $PN[i, 1]$ is the outermost bracket of the parse. $P[i, 2]$ is the next bracket under it, and so on. $P[i, NP[i]]$ is the "innermost" bracket, namely the one covering the last parsed symbol.

The algorithm for parsing the "next" (q)th symbol in the object string (call it $O[q]$) is:

```

t := N + 1;
for i := 1 step 1 until N do
begin
  for j := NP[i] step -1 until 1 do
  begin
    for k := 1 step 1 until NC[O[q]] do
    begin
      SW := true;
      l := PS[i, j];
      L2: if CN[k, O[q]] = SN[l] then
      begin
        COPY PARSE (i, j);
        L1: if CP[k] ≠ 0 then
        begin
          j = j + 1;
          PN[t, j] := CN[k];
          PS[t, j] := CS[k];
          PI[t, j] := q;
          k := CP[k];
          go to L1 end;
        t := t + 1 end;
        if l = 1 then SW := false;
        if SA[l] ≠ 0 then begin l := SA[l]; go to L2 end;
      end;
      if SW then go to L3 end;
    L3: end
  end
end

```

The procedure COPYPARSE is defined as follows:

```

procedure COPYPARSE (i, j); value i, j;
begin
  for u := 1 step 1 until j do
    begin
      PN[t, u] := PN[i, u];
      PS[t, u] := PS[i, u];
      PI[t, u] := PI[i, u]; end;
      PS[t, u] := SS[PS[t, u]]
    for u = j + 1 step 1 until NP[i] do
      Output appropriate information about PN[i, u] etc. Such out-
      puts specify the final parse.
    end
  end

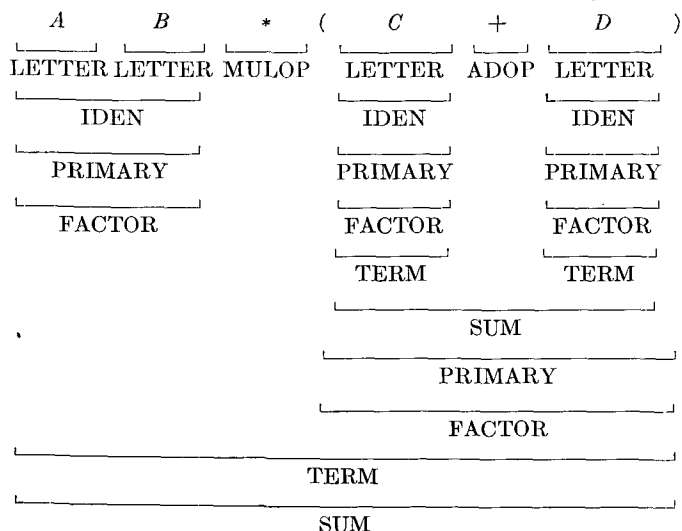
```

After executing these program steps, the parses of the object string lie in *NP*[*N* + 1], *NP*[*N* + 2] ... They are then moved to *NP*[1], *NP*[2] ... and the process is repeated for the next symbol in the object string.

As an example of the parsing operation we give a blow-by-blow description of the parse of

$$AB * (C + D)$$

according to the syntax of our example. The final parse is:



The chain for *A* is

<i>i</i>	<i>CN</i> [<i>i, 'A'</i>]	<i>CS</i> [<i>i, 'A'</i>]	<i>CP</i> [<i>i, 'A'</i>]
1	A	0	0
2	letter	1	1
3	iden	2	2
4	primary	4	3
5	factor	6	4
6	term	8	5
7	sum	10	6

The complete syntax tree is:

<i>i</i>	<i>SN</i> [<i>i</i>]	<i>SA</i> [<i>i</i>]	<i>SS</i> [<i>i</i>]
1	null	0	0
2	letter	3	2
3	digit	1	2
4	sum	0	5
5)	0	1
6	↑	1	7
7	primary	0	6
8	mulop	1	9
9	factor	0	8
10	adop	1	11
11	term	0	10
12	term	0	1

The parse (there is only one at all times for this example) is (we abbreviate the syntactic names by their first letter):

	<i>PN</i> [<i>i</i>], <i>PS</i> [<i>i</i>]									
<i>O</i> / <i>i</i>	10	9	8	7	6	5	4	3	2	1
A					L, 1	1, 2	P, 1	F, 6	T, 8	S, 10
B					L, 1	I, 2	P, 1	F, 6	T, 8	S, 10
*								M, 1	T, 9	S, 10
(P, 4	F, 6	T, 8	S, 10
C	L, 1	I, 2	P, 1	F, 6	T, 8	S, 10	P, 5	F, 6	T, 8	S, 10
+					A, 12	S, 11	P, 5	F, 6	T, 8	S, 10
D	L, 1	I, 2	P, 1	F, 6	T, 8	S, 10	P, 5	F, 6	T, 8	S, 10
)							P, 1	F, 6	T, 8	S, 10

The output of the program is simply a list of brackets equivalent to the pictorial parse diagram given earlier.

Error Correction Algorithm

An error in the object string will cause all parses to disappear at or shortly after the error. In this event the following actions are taken:

1. A list is compiled of all the syntactic elements or basic symbols which might be called for after the error point. The list consists of all elements of *SN* named by the syntax pointers of all brackets in all parses (just before the error point) and all successors and alternates of these *SN* elements.

2. The symbols at and after the error point are examined one by one and discarded until one is found which
 - a. occurs on the list of 1, or
 - b. has an element on its chain which occurs on the list of 1.

3. The bracket from 1 which is selected in 2 is examined in relation to the parses to determine a string of basic symbols which, when inserted at the error point will allow the parse to continue at least one symbol past the inserted string.

4. The string of 3 is inserted into the object string at the error point and the parse is continued. The parse is forced to cover the complete input string by initializing the parse with a "program" bracket which requires a special symbol (to be inserted at the end of input string) for its termination.

The pilot model used to verify these algorithms used the syntax productions of Figure 1 to produce the parse and error diagnostic shown in Figure 2.

An interesting side effect of the parse algorithm is that ambiguous strings for a set of productions are easily detected since they will cause the occurrence of two or more *identical* parses in *PN* at the end of the ambiguous string. Such occurrences cause all but one of the parses to be dropped and the printing of appropriate diagnostic information.

The most important application of the error correcting parse algorithm is to compiler construction. The error correction feature will allow compilers using this technique to compile and run an error ridden program to obtain a

Applications

The most important application of the error correcting parse algorithm is to compiler construction. The error correction feature will allow compilers using this technique to compile and run an error ridden program to obtain a

SYNTAX RULES

1. METAVARIABLES ARE ENCLOSED IN PARENTHESES.
2. NO VERTICAL BAR ALLOWED.
3. USE + AND - FOR LEFT AND RIGHT BRACES RESPECTIVELY.
4. THE FOLLOWING RULES PROVIDE FOR INSERTING BASIC SYMBOLS {}+*

USE 'L FOR (
 USE 'R FOR)
 USE 'P FOR '
 USE 'A FOR +
 USE 'S FOR -

5. ASSIGNMENTS ARE TO THE RIGHT RATHER THAN TO THE LEFT.
 I.E. (A){B}=(C) MEANS AN A CONCATENATED WITH A B FORMS A C.

PRODUCTIONS FOR FIGURE 2.

```

(SL) =(PG)
A=(LT)
B=(LT)
C=(LT)
D=(LT)
E=(LT)
F=(LT)
G=(LT)
H=(LT)
I=(LT)
J=(LT)
K=(LT)
L=(LT)
M=(LT)
N=(LT)
O=(LT)
P=(LT)
Q=(LT)
R=(LT)
S=(LT)
T=(LT)
U=(LT)
V=(LT)
W=(LT)
X=(LT)
Y=(LT)
Z=(LT)
'A=(AO)
'S=(AO)
*=(MO)
/=(MO)
(LT)+(LT)-={PR}
'L(SU)'R={PR}
(PR)+(MO)(PR)-={TM}
(TM)+(AO)(TM)-={SU}
(LT)+(LT)-={SU}={ST}
(ST)+; (ST)-={SL}

```

FIG. 1

maximum of diagnostic information in one try on a machine. The success of the CORC compiler testifies for the merits of this mode of operation. We reiterate the earlier statement that constructing a good compiler is still far from being a trivial task; output code optimization and "self-defining" or declarative languages are just two areas which still present difficulties in compiler construction which are not solved by (indeed are partly outside the scope of) automatic parsing techniques. The error correcting parse will, however, remove some of the burdens of programming a good compiler.

A second area of application which may have some importance in the future is in the area of pattern recognition. One of the biggest problems in pattern recognition devices is their lack of ability to capitalize as the human reader does on the wealth of contextual information contained in many patterns of interest. A combination of the error correcting parse and a pattern recognizing device which, for example, might offer several interpretations of a pattern and weight for each, might produce an effective device for reading and interpreting names on forms, information in journals and the like. At the very least, we might hope to allow a programmer to present his hand-

INPUT STRING

;(RE=-V2))(*XM;*X=A*F;X=-HT.(R)*ST;EN

DIAGNOSTICS

```

IN COL 01 OF CARD 001 REPLACES ;(
IN COL 06 OF CARD 001 REPLACES -
IN COL 08 OF CARD 001 + REPLACES 21)
IN COL 12 OF CARD 001 I) REPLACES
IN COL 16 OF CARD 001 REPLACES +
IN COL 24 OF CARD 001 = REPLACES -+
IN COL 28 OF CARD 001 + REPLACES .
IN COL 01 OF CARD 002 =I REPLACES

```

PARSE

R 0,0	LSSP
E 0,0	LSSP
= 0,0	SSP
V 0,0	LPTSSSP
+ 0,0	ASSSP
(0,0	PTSSSP
I 0,0	LPTSPTSSSP
) 0,0	PTSSSP
* 0,0	MTSSSP
X 0,0	LPTSSSP
M 0,0	LPTSSSP
; 0,0	SP
X 0,0	LSSP
= 0,0	SSP
A 0,0	LPTSSSP
+ 0,0	ASSSP
F 0,0	LPTSSSP
; 0,0	SP
X 0,0	LSSP
= 0,0	SSP
H 0,0	LPTSSSP
T 0,0	LPTSSSP
+ 0,0	ASSSP
(0,0	PTSSSP
R 0,0	LPTSPTSSSP
) 0,0	PTSSSP
* 0,0	MTSSSP
S 0,0	LPTSSSP
T 0,0	LPTSSSP
; 0,0	SP
E 0,0	LSSP
N 0,0	LSSP
SSP	SSP
I 0,0	LPTSSSP

FIG. 2

written XGOL program to the computer thus avoiding the very serious restrictions of card punch and typewriter character sets.

REFERENCES

1. IRONS, E. T. A syntax directed compiler for ALGOL 60. *Comm. ACM* 4 (1961), 51-55.
2. ——— Towards more versatile mechanical translators. To be published.
3. ——— The structure and use of the syntax directed compiler. *Ann. Rev. in Autom. Programming*, 3, 207-228.
4. PAUL, M. A General Processor for Certain Formal Languages. *Symbol Languages in Data Processing*. Gordon and Breach, London 1962, 65-74.
5. EICKEL, J., PAUL, M., BAUER, F. L., SAMUELSON, K. A syntax controlled generator of formal language processor. Institut für Angew. Mat. der Univ. Mainz. Sept., 1962.
6. BROOKER, R. A., MACCALLUM, I., MORRIS, D., ROHL, J. S. The compiler compiler. *Ann. Rev. Autom. Programming* 3, 229-271.
7. BACKUS, J. W. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conf. Proc. Internat. Conf. Inform. Process., UNESCO, (June 1959), 125-132.
8. NAUR, PETER (ed.). Report on the algorithmic language ALGOL 60. *Comm. ACM* 3 (1960) 299-314.

