Title	An error recovery method for programming languages without separators between statements
Sub Title	
Author	Nakanishi, Masakazu Ono, Yoshio
Publisher	慶応義塾大学工学部
Publication year	1979
Jtitle	Keio engineering reports Vol.32, No.1 (1979. 3) ,p.1- 5
JaLC DOI	
Abstract	A new method to handle syntax errors in the programming languages without separators between statements is proposed. By using this method, we can minimize the printing of causeless error messages. It does not backtrack and is suitable for small computers.
Notes	
Genre	Departmental Bulletin Paper
URL	https://koara.lib.keio.ac.jp/xoonips/modules/xoonips/detail.php?koara_id=KO50001004-00320001-0001

KEIO ENGINEERING REPORTS
VOL. 32, NO. 1. pp. 1-5, 1979

AN ERROR RECOVERY METHOD FOR PROGRAMMING LANGUAGES WITHOUT SEPARATORS BETWEEN STATEMENTS*

MASAKAZU NAKANISHI AND YOSHIO OHNO**

Dept. of Mathematics, Keio University, Hiyoshi, Yokohama 223, Japan

(Received December, 18, 1977)

ABSTRACT

A new method to handle syntax errors in the programming languages without separators between statements is proposed. By using this method, we can minimize the printing of causeless error messages. It does not backtrack and is suitable for small computers.

1. Introduction

When a line-free language such as ALGOL is automatically analyzed by a processor, it is difficult to define how to omit characters following the token in which a syntax error is detected. The simplest method is one to omit remaining characters until a delimitor is read. For example, ",", ")", ";", brackets etc. are delimitors in ALGOL. Let us consider the following compound statement which includes two illegal statements.

begin

i : =a+b↑ ÷ −c ;

a : =b×c)

end

When the error beginning with the character "÷" in the first statement of the above compound is detected, "÷", "−" and "c" are omitted and the processing of the first statement is finished. But the processing of this compound is not yet finished and the remaining statement must be analyzed. The delimitor is used not only for separating statements but also for signalling an end to omitting characters when a syntax error is detected. Therefore, the delimitor takes an

* The abstract has been published in *Keio Mathematical Seminar Report*, 2, 1977. This is partly supported by The Matsunaga Research Grant.

** Keio Institute of Information Science.

important part for the processor.

SIMPL* is a programming language which has no delimitors to separate its statements. The analyzer can detect the end of a statement by reading the first token of the next statement. But it cannot simply omit characters if an error is detected, because there is no delimitor to signify an end to the omitting process.

This paper describes an algorithm to handle syntax errors and to determine which characters have to be omitted after an error is detected for a programming language without delimitors. It is a simple algorithm which does not backtrack, and it is suitable for the SIMPL compiler when used by small computers.

2. Syntax Errors of SIMPL

In the SIMPL compiler, the first token of the next statement is used as the separator of statements.

```
IF X=Y THEN
    A : =B-C*(D-E+F)/G
    H : =X-Y*F
END
```

In the above example, H and END have the roles of separators. If "+" in the first assignment statement is missing, i.e., A : =B-C*(D-E F)/G, the compiler usually stops analyzing at token "F", but "F" is not the first token of the next statement. Thus, "F)/G" must be omitted. In the second assignment statement if a "/" is detected following the "*", i.e., H : =X-Y*/F, then the "*/F" or the "/" must be omitted.

A primary is the basic unit in syntax error detection, and the primaries in SIMPL are shown in the following:

i) a constant or a variable; e.g., 3.14159, VAR.
ii) a subscripted variable; e.g., A(I+J).
iii) a function designator; e.g., FUNC(X, Y-Z).
iv) an expression enclosed in parentheses; e.g., (X+Y-Z).
v) a primary with a partial word designator; e.g., (X+Y-Z) [15, I+2].

In the last primary, 15 is the starting bit number and I+2 is the length of the bit string.

The SIMPL syntax errors can be classified as follows:

E1. The position of a right parenthesis is not correct, or there is no right parenthesis.
E2. A primary is beginning with a delimitor (or a reserved word).
E3. A primary is beginning with an operator.
E4. An element in the argument list is not separated by a comma, or there is no right parenthesis.

---

* SIMPL has been designed and implemented at University of Maryland for a tool of system implementation (BASILI, *et al.* 1975).

Each of the above errors is shown in the following statements respectively.

E1. A:=B−C*(D−E F)/G Z:=A+1 .....
A:=B−C*(D−E, F)/G Z:=A+1 .....
E2. A:=B−C* IF X=0 THEN .....
A:=END
E3. A:=B−C*/D
E4. A:=F(A+B, C−D E)
A:=F(A+B, C−D$E)

One way to omit characters is to omit some tokens until a ":=" or any reserved word is read. However a new and different method is proposed in the next section to reduce the characters to be omitted by inserting an operator or an operand into the erroneous expression. It can also prevent to print causeless error messages.

3. The Algorithm

A routine that classifies the statements in the source program and gives control to their analyzers is usually called a dispatcher. In this algorithm, the dispatcher uses a word named SYNERR to indicate that some errors are detected in a statement. It is set by the codes of the errors and the token which causes the error, and it is reset before the dispatcher calls each analyzer of a statement.

The names of the rules to handle syntax errors correspond to the classification of errors described in the previous section. If the error E1 is detected, the rule E1 is applied. The symbols Φ and ϕ in the following rules are called the *imaginative operand* and the *imaginative operator* respectively. They do not exist in the source program but are assumed by the processor in order to recover syntax errors. The imaginative operator can be regarded as the operator with lowest priority.

Rule E1:

i) If an identifier or a constant is detected instead of a right parenthesis, the process of analyzing the expression is repeated. The effect of this action corresponds to the insertion of an imaginative operator before the identifier or the constant. Then an error code is set in SYNERR.
Example: A:=B−C*(D−E F)/G is analyzed as if it is A:=B−C*(D−EϕF)/G.

ii) If a delimitor (or a reserved word) is detected, the process of scanning the expression is ended and an error code is set in SYNERR.
Example: A:=B−C*(D−E IF X=0 ... is analyzed as if it is A:=B−C*(D−E) IF X=0 ... .

iii) ":=" is the only operator which can be detected in E1, because the other operators must be included in the expression which has already been analyzed. In this case, the scanning of the expression before the ":=" is ended, an error code is set in SYNERR, and then the expression after the ":=" is analyzed.
Example: A:=B−(C*(D−E)/G:=G+1 is analyzed as if it is A:=B−(C*(D−E)/G) Φ:=G+1.

iv) If a special character (not the operator) is detected, it is removed. Then an error code is set in SYNERR and the rule E1 is applied again.
Example: A:=B−C*(D−E, F)/G is analyzed as same as the example of i) because in the first application of rule E1, the comma is removed, and then in the second application of E1, an imaginative operator is assumed before "F" by i).

Rule E2:

An error code is set in SYNERR. The new token is never read if this error is detected in the primary analyzer. It means that there exists a primary that consists of the empty string, or the imaginative operand.

Example: A:=B−C* IF X=0... is analyzed as if it is A:=B−C*ϕ IF X=0... .

Rule E3:

Rule E3 is the same as E2. In this case, an imaginative operand is assumed between operators.

Example: A:=B−C*/D is analyzed as if it is A:B−C*ϕ/D.

Rule E4:

This is similar to the rule E1 except for a few points.

i) If an identifier or a constant is detected, a comma is assumed instead of some blanks. A warning message rather than an error message is printed.
Example: A:=F(A+B, C−D E) is compiled the same as A:=F(A+B, C−D, E).

ii) If a delimitor is detected, the right parenthesis is assumed. That is, the primary analyzer ends the scanning of this designator and an error code is set in SYNERR.
Example: A:=F(A+B, C−D IF A=B THEN... is analyzed as if it is A:=F(A+B, C−D) IF A=B THEN... .

iii) If a ":=" is detected, the right parenthesis is assumed, and an error code is set in SYNERR. Then the rule E3 is applied.
Example: A:=F(A+B:=A+1 is analyzed as if it is A:=F(A+B) ϕ:=A+1.

iv) If a special character is detected, it is removed and an error code is set in SYNERR. Then the rule E4 is applied again.
Example: The last example of the previous section, i.e., A:=F(A+B, C−D$E), is analyzed as if it is A:F(A+B, C−D, E) because in the second application of rule E4, a comma is assumed before "E" by i).

The error message is printed when a statement has been completely compiled. If an error code is set in SYNERR, the message is constructed by the information in SYNERR. In this algorithm, two or more errors may be detected in a statement. For A:=B−C*(D−E F:=F+1, the case i) of rule E1 is applied first and the statement is analyzed as if it is A:=B−C*(D−Eϕ F:=F+1. When the second ":=" is encountered in the analysis of this statement, the case ii) of the rule E1 is applied. Two error codes are set in SYNERR. The error message is generated by the several codes included in SYNERR. For the above example, the message "A right parenthesis is missing in the expression" may be printed. On the other hand, for A:=B−C*(D−E F) G:=G+1, "An operator is missing" is printed, because only the case i) of E1 has been found.

The dispatcher may find the statement beginning with ": =" when error codes have been set in SYNERR during compilation of the last statement. This can be considered the assignment statement with the imaginative operand in the left hand part to it. Similarly the illegal statement which does not begin with an identifier nor a reserved word is detected as an error by the dispatcher. The error message is printed and illegal tokens are omitted until an identifier or a reserved word is found. While error codes are set in SYNERR, the compiler must be inhibited from detecting semantic errors in order to suppress inadequate error messages.

4. Conclusion

It is not desirable to have many causeless error messages printed for only one syntax error. At the present time the 1108 SIMPL compiler prints 6 error messages for the statement "A: =B+C*(D−E F+G)/2". On the other hand, if after detecting one error and skipping the following tokens, that is to say, not checking for other errors, often one must recompile ones program many times to finally get rid of all the errors. This also is undesirable. ("A: =B C*(D−E*F+G) 2" has two errors which would take two compilations to detect them both.)

There are many reports (e. g., WILCOX, *et al.* 1976) about error recovery methods and almost all of them use the backtracking technique. They are often complicated and use large amounts of memory space and time. In compilers for small computers, so much work on recovering errors cannot be spent. Therefore, we cannot help but to adopt the simple method of error recovery.

This method is now built into the SIMPL implemented on a PDP-11 by K. MOCHIZUKI and K. TATSUMI. Total size of the SIMPL processor is about 20K words, and the error recovery program occupies about 0.5K words in it.

REFERENCES

[1] V. R. BASILI and A. J. TURNER (1975): *SIMPL-T: A Structured Programming Language*, Computer Science Center, University of Maryland, Computer CN-14.1.

[2] T. R. WILCOX, A. M. DAVIS and M. H. TINDALL, *The Design and Implementation of a Table Driven, Interactive Diagnostic Programming System*, 1976, Communications of the ACM, **19**, 609-616.