Compile-time error diagnostics in syntax-directed compilers

C. J. Burgess

Computer Science Department, School of Mathematics, University of Bristol, University Walk Bristol

This paper shows that it is possible to formally specify the compile-time error diagnostics provided by a syntax-directed compiler and investigates the extent to which these diagnostics can be automatically inserted into a given grammar.

(Received March 1972)

1. Introduction

One of the major tasks in computing is the production of a correctly working program. The total amount of programming time required for any particular program will be very dependent upon the quality of the diagnostics provided by the computer system. The poor quality provided by some systems has already been well illustrated, e.g. Barron (1971). It would appear that very little effort is devoted, when designing and implementing systems, to helping the user who has an incorrect program, although there are exceptions, e.g. using LISP (Teitelman, Bobrow, Hartley, and Murphy, 1971).

It would also be advantageous if the diagnostics provided could be standardised for any particular programming language. This paper concentrates on the problem of providing good diagnostics during the syntax-analysis phase of compilation. It shows how the diagnostics provided can be formally specified when defining the grammar of the language and the combined definition used directly to drive the syntax-analysis stage of the compiler. This means that for a particular set of grammars, determined only by the analysis algorithm, the error diagnostics can be regarded as part of the definition of the language.

The experiments and results referred to in the paper were obtained using a simple top-down, left to right scan analysis algorithm which allowed back-tracking. This method only imposes a few constraints upon the grammar, the major one being no left recursive definitions of syntactic categories, but for most practical programming languages these restrictions can be overcome by simple transformations of the grammar.

Finally the paper shows that for a more restricted set of grammars, viz. left-factored (LF), the process of inserting diagnostics into the given grammar can be partially automated.

2. Treatment of compile-time errors

It is convenient to divide the errors that can be detected in a program at compile-time into two main groups which will be termed syntax and semantic errors respectively. It is difficult to define precisely the boundary between these two groups to satisfy everyone. This paper will use the term syntax error to refer to a failure to match the program against the syntax-specification as expressed using BNF, and semantic error to refer to all other compile-time errors, of which the majority will be errors due to the interpretation or meaning attached to the symbols.

Examples (using ALGOL 60)

1. Syntax errors

```
    J := *3;
          ↑
    Incorrect arithmetic expression
    I := (I + J) (K + L)
                ↑
    Missing operator
```

2. Semantic errors

```
  begin
    integer I, J;
    real A;
    procedure P(A, B);
    real A, B;
    B := SQRT(LOG(A));
    ....
    ....
    goto I;
         ↑
    Incorrect label
    ....
    ....
    B := P(A, I);
              ↑
    Parameter of impermissible type
    ....
  end;
```

This paper will concentrate on syntax error messages although semantic errors can be treated in a similar way.

2.1. *Error message specification*

Since syntax errors are directly related to the form of the syntactic constructions, the most suitable place to formally specify the messages is in the constructions themselves. The syntax specification is expressed using BNF and the error messages are introduced by defining a new component which will be termed an error category. Two new metalinguistic symbols are also introduced, viz. opening and closing string quote represented by " and " respectively, and an error category is then written as an error message enclosed in string quotes, e.g.

⟨digit⟩ ::= 0|1|2|3 . . . |7|8|9|"Missing digit"

These error categories can occur anywhere on the right-hand side of syntactic constructions and when the analyser reaches that component the error message will be printed out. Considering the above example, if an error has definitely occurred when the syntactic category ⟨digit⟩ cannot be successfully matched against the input string, then an appropriate error message, in this case "Missing digit" is required every time ⟨digit⟩ is unsuccessfully matched.

A more complicated example is one possible construction for the syntax of a for statement in ALGOL 60, where ⟨forlist⟩ represents the list of all possible elements in the control section of the for statement.

⟨for statement⟩ ::= **for** ⟨identifier⟩ := ⟨forlist⟩ **do**
  | **for** ⟨identifier⟩ := ⟨forlist⟩ "Missing do"
  | **for** ⟨identifier⟩ := "Incorrect list of elements in a for statement"
  | **for** ⟨identifier⟩ "Missing assignment sign"
  | **for** "Invalid control identifier"

It is obvious from the layout of the constructions precisely at which point in the syntax the analysis failed for any given error message enabling an accurate detailed error message to be given to the user. Also, since the syntactic constructions are scanned one at a time from left to right, no time is spent using constructions containing error messages in the above syntax unless the correct syntax has failed, i.e. there is definitely an error present. Therefore an increased number of error messages providing better diagnostics will not alter the speed of compilation of correct programs. Also the extra requirement in store size is very little more than the extra space required for the storage of the actual error messages.

2.2. *Error detection*

The majority of syntax-directed compilers will detect syntax errors within one line of their actual occurrence but frequently the point reached in the scanning of the input string when the error is detected is at least several characters past the actual error. Using the method of error specification just outlined it is possible to locate the error exactly for the majority of errors.

For example, use the same construction for the for statement in ALGOL 60 as in the previous example, to analyse the input string

for $A + B, C$ **do**

The analysis algorithm will match the **for** and ⟨identifier⟩ of the input string against the first alternative definition of ⟨for statement⟩ but will fail to match the assignment sign, viz. :=, against the +. It will therefore go on to the fourth alternative definition and reach an error category with the input string pointer pointing to the + sign. Thus the following output would be produced for the user:

```
for A + B, C do
      ↑
Missing assignment sign
```

Similarly:

```
for A := B step C until D if A = E then . . . .
                          ↑
Missing do
for := C, D do
    ↑
Missing control identifier
for A := @, C do
         ↑
Incorrect list of elements in a for statement
```

Occasionally it is necessary for the compiler writer to have even finer control over the exact position of the pointer. This is done by including after the error message in the syntax specification, a number preceded by an asterisk which indicates the number of basic symbols the pointer should be moved back before being output.

For example, part of a possible syntax for an arithmetic expression:

```
⟨arithmetic expression⟩ ::= ⟨term⟩⟨operator⟩⟨arithmetic
                                                expression⟩
                          | ⟨term⟩ ( "Missing operator*1"
                          | ⟨term⟩
```

would produce diagnostics of the form:

```
I := (I + J) (K + L)
            ↑
Missing operator
```

whereas without the *1 the pointer would have been under the *K*.

This facility increases the detail that can be provided by error messages as it allows symbols to be matched just prior to an error category which are not part of a correct program, e.g. the (just after ⟨term⟩ in the above example.

The same facility can also be used to reduce the number of syntactic categories required in the syntax specification, for example, the need in ALGOL 60 to have separate categories for conditional and unconditional source statements

```
⟨source statement⟩ ::= ⟨if clause⟩ if "if not allowed
                                immediately after a then *1"
                     |⟨if clause⟩ ⟨source statement⟩
⟨if clause⟩ ::= if ⟨boolean expression⟩ then
```

2.3. *Error recovery*

The ability to check the rest of a program once an error has been detected is a difficult problem for the majority of compilers. Nearly every practical compiler employs some form of *ad hoc* error recovery technique which is extremely language dependent and sometimes also based on experience gained from users of the most frequently occurring errors. The major difficulty is trying to strike the right balance between detecting all subsequent errors after the first, and not producing any spurious error messages, i.e. error messages that do not correspond to actual errors but are produced purely as a result of previous errors. It is not always clear what should be treated as a separate error and what should be ignored as purely a result of a previous error, e.g. a single error in the block structure of an ALGOL 60 program often causes a large number of errors associated with scopes of identifiers. Should all these errors have explicit error messages?

In an attempt to develop a fairly language independent method of error recovery the following techniques were used with the top-down analysis algorithm.

1. If the language has a natural statement terminator or set of terminators, e.g. in ALGOL 60, ; and **end** , ignoring **else** as this may be part of a conditional expression and not a conditional statement, then the input string pointer was wound on to the next statement terminator and the analysis algorithm continued from that partially matched syntactic construction which expected that statement terminator as its next character, e.g. suppose the following construction was part of the syntax of a language which used a semicolon as a statement terminator

   ```
   ⟨list of statements⟩ ::= ⟨statement⟩;⟨list of statements⟩
                          |⟨statement⟩
   ```

   and the analysis algorithm found the following error when analysing

   ```
   A := B* − C; B := D;
            ↑
   Incorrect operand.
   ```

   the input pointer is then wound on to point to the semicolon and the analysis proceeds by unwinding the stack of partially matched syntactic constructions until it finds the most recent use of ⟨list of statements⟩, and then proceeds to match the semicolon followed by $B := D$; against a further call of ⟨list of statements⟩.

2. If the language has no natural statement terminator or the stack of partially matched syntactic constructions does not include any expecting a statement terminator as the next character, then a more general and slower recovery algorithm is used. This involves constructing a list of those characters which are expected as the next character in partially matched syntactic constructions and then searching the input string for any one of these and continuing from the appropriate syntactic construction.

Method (2) is highly dependent upon the particular analysis algorithm used and relies upon using individual characters to uniquely identify the context, and thus guide the analysis algorithm on to the right path. Since in most languages, both the digits and letters are used in a number of different contexts, these characters are omitted when constructing the list of possible characters unless they constitute part of a basic word. In practice this method, when used alone, gave very poor results since individual characters often occur in several different contexts, but it did provide a useful addition to method (1) when analysing ALGOL 60 programs for those few cases where the current line did not contain a statement terminator. An example of this method can be found in Burgess (1971).

If satisfactory error recovery is to be achieved for a language with no natural statement terminator, then a far better and probably more time and space consuming method would be required, which would most likely also be very language dependent.

3. Construction of the syntax specification

The main problem which remains to be solved is, given the syntax specification of syntactically correct programs, where should the additional alternative definitions involving error categories be inserted. The major difficulty is associated with the insertion of error categories so as to avoid the rejection of syntactically correct programs due to the premature matching of a program against a construction containing an error category.

For example, consider the syntactic constructions

$$\langle X \rangle ::= \langle A \rangle \mid \langle B \rangle$$
$$\langle A \rangle ::= a\,b \mid a \text{“ERROR”}$$
$$\langle B \rangle ::= a\,c$$

Every time this syntax is used to match the syntactic category $\langle X \rangle$ against the input string *ac* an error message will be produced due to the second alternative definition of $\langle A \rangle$. This cannot be deduced by examining just the definition of $\langle A \rangle$ but only becomes clear when considering possible alternatives to the match of $\langle A \rangle$ as a component within the definition of other categories.

This example could easily be rewritten to avoid the problem but in general this is not possible.

Definition

The *insertion* of an *error category* will be said to be *permitted* if it does not cause the premature match of any syntactically correct program.

Conversely, the *insertion* of an *error category* will be said to be *prohibited* if it does cause the premature match of any syntactically correct programs.

It is difficult, or impossible, to devise a method for a general syntax of determining all the permitted insertions of error categories. The remainder of this paper will be devoted to two particular approaches to the problem and the results achieved, viz.:

1. Error message insertion into a general syntax using experimental testing to eliminate prohibited insertions.
2. Automatic error message insertion into a particular class of grammars with a guarantee that all insertions are permitted.

3.1. *Error message insertion into a general syntax*

The general approach adopted was to insert error messages into all the appropriate places in the syntax (defined later) and then using a few simple guidelines and subsequently test programs, remove those insertions which are prohibited.

The syntactic constructions are stored so that the time required by the analysis algorithm is minimised. In particular, if two alternative definitions of the same syntactic category start with one or more identical components, then no back-tracking occurs involving these components unless both the alternative definitions fail, e.g.

$$\begin{aligned}\langle X \rangle ::= &\ \langle A \rangle \langle B \rangle\\ &|\langle A \rangle \langle C \rangle\end{aligned}$$

will not involve any back-tracking associated with the match of $\langle A \rangle$ unless both the definitions fail to match. In this example then, $\langle C \rangle$ can be considered to be an *alternative component* to $\langle B \rangle$, i.e. if $\langle B \rangle$ fails to match then an attempt will be made to match $\langle C \rangle$.

There are then two types of position in a syntax which are possible places for permitted error message insertion which will not slow down the analysis of syntactically correct programs, viz. (i) As an alternative component to any other component that has not a direct alternative already specified, including the null string, e.g. in the syntax

$$\begin{aligned}\langle X \rangle ::= &\ \langle A \rangle \langle B \rangle \langle C \rangle\\ &|\langle A \rangle \langle D \rangle\\ &|\langle A \rangle\end{aligned}$$

$\langle C \rangle$ has no direct alternative component, but $\langle B \rangle$ has $\langle D \rangle$ and $\langle D \rangle$ has the null string. Thus only one error message can be inserted, viz.:

$$\begin{aligned}\langle X \rangle ::= &\ \langle A \rangle \langle B \rangle \langle C \rangle\\ &|\langle A \rangle \langle B \rangle \text{“ERROR”}\\ &|\langle A \rangle \langle D \rangle\\ &|\langle A \rangle\end{aligned}$$

(ii) As an extension of (i), an error message could be inserted as the only component in the last alternative definition of a syntactic category, e.g.

$$\begin{aligned}\langle Y \rangle ::= &\ \langle A \rangle\\ &|\langle B \rangle\\ &|\text{“ERROR”}\end{aligned}$$

However, this has the same effect as an insertion of type (i) after every occurrence of $\langle Y \rangle$ as a component in the definitions of other syntactic categories, except for the largest category, viz. ⟨program⟩. Thus, with the exception of ⟨program⟩ this type of insertion is covered by type (i) and will therefore be ignored until the final stages of the insertion process.

The major problem then reduces to deciding which of the type (i) insertions are permitted. If the syntax never involves back-tracking then the problem is solved since it can be shown that all type (i) insertions are permitted. This is the basis of the automatic error message insertion described in the next section.

For a general syntax there is no easy way of determining all those constructions which are involved in back-tracking during the analysis of syntactically correct programs. One possible approach is to eliminate all error categories associated with the definition of syntactic categories which have direct alternative components, e.g. considering the syntactic construction

$$\begin{aligned}\langle X \rangle ::= &\ \langle A \rangle \langle B \rangle\\ &|\langle A \rangle \langle C \rangle\end{aligned}$$

this means not only removing any error categories used in the definition of $\langle B \rangle$ but also any used in the definition of other components used in the definition of $\langle B \rangle$. Applying this criterion to the whole syntax of a practical programming language eliminates nearly all the error categories, particularly where recursive or mutually recursive definitions are used.

A less severe criterion, but one which requires more computation involves constructing a matrix (one row for every syntactic category), of all the possible characters which can start a string which will match that syntactic category. This is effectively a one character look-ahead matrix which may anyway be incorporated as part of the analysis algorithm. Then wherever direct alternative components exist, this matrix is used to determine whether the strings which could match the components have any starting characters in common. If no such characters exist, then no premature matching by error categories used in the definition of the first component can occur. If there are common characters then back-tracking can definitely occur involving the first component, and the error categories used in its definition must be removed.

One other criterion which can be used is based on a familiarity with the programming language concerned. If it is known that a given symbol is only used in one particular context, or alternatively after a group of symbols or some semantic check that the context is unique, then error insertions subsequent to that point in the syntactic constructions can remain.

For example, suppose after matching a ⟨term⟩ and either a plus or a minus sign, then the context must be an arithmetic expression, then a construction of the following type could be used without fear of premature matching.

```
⟨arithmetic expression⟩ ::= ⟨term⟩ + ⟨arithmetic expression⟩
                          |⟨term⟩ + "Incorrect operand"
                          |⟨term⟩ − ⟨arithmetic expression⟩
                          |⟨term⟩ − "Incorrect operand"
```

Any other error categories which may be prohibited have to be found by running test programs but this is the price of allowing a completely general syntax and still providing reasonable error diagnostics.

These methods were used in the implementation of a syntax checker for ALGOL 60 which also incorporated semantic checking, and a working syntax was produced with an adequate number of error categories left in the syntax. The method was also applied to a BNF definition of SNOBOL 4, but although this worked, proportionately fewer error categories were left in the syntax as several characters in the language are used in a large number of different contexts, in particular the space character (Burgess, 1971).

3.2. *Automatic insertion of error messages*

The major characteristic of a grammar which makes the insertion of error messages difficult is the need for back-tracking during parsing. Some work has already been done on grammars which do not require back-tracking during parsing. In particular, Foster describes an algorithm called SID which attempts to transform any given grammar into the required form (Foster, 1968). A detailed theory of this type of grammar, termed a left-factored or LF grammar, has been given by Wood (Wood, 1969). The terminology used in this section is based on that used by Wood, but the relevant sections are repeated here for completeness together with some extensions. The notation uses productions rather than BNF for syntactic constructions as these are easier to manipulate but any grammar including error categories can easily be written in either form.

Terminology

A normal context-free grammar G can be defined as a 4-tuple

$$G = (I, T, S, P),$$

where I is a finite set of intermediate symbols which appear on the left of productions in P, T is a finite set of terminal symbols which appear only in the right of productions in P (excluding the null string ϵ), S is the sentence symbol (same role as ⟨program⟩) where $S \in I$, P is a finite set of productions of the form $X \rightarrow q$, $X \in I$, $q \in (I \cup T)^*$ where A^* represents the set of all strings over the alphabet A including the null string.

In future IT, T' and IT' will represent $I \cup T$, $T \cup \{\epsilon\}$, and $I \cup T \cup \{\epsilon\}$ respectively, and G will imply the 4-tuple (I, T, S, P) unless explicitly stated otherwise.

A *sentence* of a grammar G is now defined as any string s, $s \in A^*$, for which a left-sentential derivation exists.

A *correct sentence* in a grammar G is a sentence for which a unique left-sentential derivation exists that does not use any productions which contain error categories, i.e. the sentence has a unique parse and no error messages will be produced.

Conversely an *incorrect sentence* in a grammar G may have more than one left-sentential derivation but each derivation will involve using a production which contains an error category, i.e. during a parse an error message will always be produced.

An *error category* is a special terminal symbol which will match any string of symbols in the alphabet A. When matched during the parsing of a sentence it will result in an error message being produced with a pointer to the first character of the string it matched (unless modified using the *n facility described in Section 2.2).

These definitions allow only the first error in an incorrect sentence to be found, the error category matching the remainder of the sentence. This is sufficient for most of the theory which ignores the practical problems of error recovery and the detection of subsequent errors.

Formal definition of the problem

Given a grammar $G = (I, T, S, P)$, we need to define a corresponding grammar G' which will have the following properties:

Property 1
All correct sentences in grammar G must also be correct sentences in grammar G'.

Property 2
Any string s, $s \in A^*$, must be a sentence in grammar G', but is only a correct sentence in G' if it is also a correct sentence in grammar G, i.e. any string has at least one parse using grammar G', but only those strings which are incorrect sentences in G will involve productions using error categories when parsed using G'.

The process of correcting the syntax of a program then becomes the editing of a sentence in G' so that it becomes a correct sentence.

Solutions to the problem for left-factored grammars

A trivial solution

The easiest way to construct G' given G, is to add one production of the form:

$$S \rightarrow \text{“Invalid sentence”}$$

i.e. one more production added to the definition of the sentence symbol S. Thus if a sentence cannot be parsed using G, eventually when parsing using G', this final production for S will be used and the error message produced.

This solution provides little or no information as to how the sentence differs from a correct sentence, i.e. information to enable a program to be corrected, and is therefore of little practical value.

Main solution

This is an attempt at a solution which will provide detailed information to enable incorrect sentences to be changed into correct sentences.

Define G' to be a 6-tuple

$$G' = (I, T, S, P, Q, E),$$

where $G = (I, T, S, P)$ is the left-factored grammar specified, E is a finite set of error categories, and Q is a finite set of productions of the form

$$X \rightarrow uk \text{ where } u \in IT'^* \text{ and } k \in E.$$

The problem then reduces to the precise definition of Q, such that G' has the required properties.

Definition of Q

There are two main subsets in Q

1. Q includes a production of the form $S \rightarrow k$, where $k \in E$ and S is the sentence symbol.

This production has the effect that for all sentences which cannot be matched using any other parse, the error message k is produced. If this was the only member of Q, then this solution corresponds to the trivial solution.

2. For all productions in P of the form:

$$X \rightarrow aY\omega,\ X \in I,\ a \in IT^*,\ Y \in IT \text{ and } \omega \in IT'^*,$$

include a production in Q of the form

$$X \rightarrow ak \text{ where } k \in E\,,$$

provided that if $Y \in I$ there exists no derivation of the null string from Y, i.e. $Y \overset{*}{\nrightarrow} \epsilon$. This last criterion is necessary to avoid premature matching occurring if correct sentences existed whose parse matched Y against the null string.

Theorem

The grammar G' defined with this set of productions Q has both properties 1 and 2 provided that before any production in Q is used the corresponding production in P has been rejected. The later clause is required since the grammar G' is ambiguous.

Proof

There are two parts to the proof

1. Grammar G' has property 1, viz. that all correct sentences in grammar G will also be correct sentences in grammar G'.

All correct sentences in G will have at least one parse in G', since the productions in G' include all the productions P in G. There will be one and only one parse which uses only the productions P, since G is left-factored and therefore unambiguous. It is therefore only necessary to show that the parse using only productions from P is the first parse.

Consider any production $X \rightarrow aY\omega$ in P, for which there exists a rule $X \rightarrow ak$ in Q.

(*a*) Since G is left-factored there cannot exist any rules of the form:

$$X \rightarrow a \text{ or } X \rightarrow aZ \text{ where } Z \in IT^*\,,$$

in P since they have common left terminal sets to the production $X \rightarrow aY\omega$.

(*b*) Since G is left-factored, no back-tracking is required during parsing.

Thus the production $X \rightarrow ak$ will only be used during the parsing if, and only if, the production $X \rightarrow aY\omega$ has been rejected. But since no back-tracking is required for the parsing of correct sentences this production will only be rejected when parsing incorrect sentences.

$$\therefore \text{ grammar } G' \text{ has property 1}\,.$$

2. Grammar G' has property 2, viz. that any string s, $s \in A^*$, must be a sentence in grammar G', but is only a correct sentence in G', if it is also a correct sentence in G.

(*a*) If a sentence is an incorrect sentence in G, then by definition it cannot be parsed using only the productions P.

(*b*) All possible sentences have at least one parse in G' since Q includes a production of the form

$$S \rightarrow k,\ k \in E$$

$\therefore$ grammar G' has property 2.

The difference between this solution and the trivial solution is that for any production that is used, the next symbol in the input string must be a member of the left terminal set for that production. If then subsequently, the production fails to match completely, there exists a production in Q with an appropriate error category. Thus the diagnostics provided are far more detailed than those produced by the trivial solution.

Thus grammar G' has both the required properties.

Error detection using G'

Using the above definition of G', it can be shown that the error pointer will always point to the first inadmissible symbol.

Proof

1. Consider the production

$$X \rightarrow ak,\ X \in I,\ a \in IT^*,\ k \in E$$

whose error category k has resulted in an error message being given. Then by definition a production

$$X \rightarrow aY\omega,\ Y \in IT,\ \omega \in IT'^*$$

has been tried and rejected.

This production is part of the left-factored grammar G, therefore a must have matched a valid string and the next symbol cannot be either Y if $Y \in T$, or a member of any of the left terminal sets of productions of the form $Y \rightarrow v$, $v \in IT^*$, if $Y \in I$, since otherwise this production would have been tried, and back-tracking is not required for left-factored grammars, in particular G. Therefore the next symbol after the string matching a cannot possibly constitute part of a valid sentence, but will be the symbol above the error pointer since it will be the first symbol of the string that matches k.

2. If, and only if, the very first symbol of a program is wrong will the production $S \rightarrow k$, $k \in E$ be used. Thus the theorem is true for the first and any subsequent symbols of a sentence.

3.3. *Results for a simple grammar*

It is rather inconvenient to write the whole syntax of a programming language as a left-factored grammar, but a very similar grammar, which will be termed LF* enables the syntax to be written more compactly and is easily transformed into a LF grammar.

Consider any two productions of the form:

$$X \rightarrow aY\omega,\ X \rightarrow aZv$$

where

$$X \in I,\quad Y, Z \in IT,\quad \omega, v \in IT'^*,\quad a \in IT^*\,.$$

These can be transformed into

$$X \rightarrow aL,\ \ L \rightarrow Y\omega,\ \ L \rightarrow Zv$$

where

$L \in I$ (L is a new intermediate symbol).

This transformation is applied to the complete grammar and if the resulting grammar is LF, then the original grammar is termed LF*.

The automatic error insertion algorithm can be applied to either the LF grammar and then if required, the grammar transformed back, or applied directly to the LF* form with slight modifications to allow for the transformation.

The following syntax which is an LF* grammar will provide an actual example. It represents a very simple language consisting of a series of arithmetic assignment statements using variables A to D, and terminated by a \$. It is specified using BNF.

```
⟨program⟩          ::= ⟨listofstatements⟩ $
⟨listofstatements⟩ ::= ⟨statement⟩; ⟨listofstatements⟩
                      |⟨statement⟩
⟨statement⟩        ::= ⟨variable⟩ = ⟨expression⟩
⟨expression⟩       ::= ⟨term⟩ + ⟨expression⟩
                      |⟨term⟩ − ⟨expression⟩
                      |⟨term⟩
⟨term⟩             ::= ⟨factor⟩ * ⟨term⟩
                      |⟨factor⟩ / ⟨term⟩
                      |⟨factor⟩
⟨factor⟩           ::= ⟨variable⟩
                      |(⟨expression⟩)
                      |⟨number⟩
⟨number⟩           ::= ⟨digit⟩ ⟨number⟩
                      |⟨digit⟩
⟨digit⟩            ::= 0|1|2|3|4|5|6|7|8|9
⟨variable⟩         ::= A|B|C|D
```

The insertion algorithm produces the following grammar:

```
⟨program⟩          ::= ⟨listofstatements⟩ $
                      |⟨listofstatements⟩ "Error number 2"
                      |"Error number 1"
⟨listofstatements⟩ ::= ⟨statement⟩; ⟨listofstatements⟩
                      |⟨statement⟩ "Error number 3"
                      |⟨statement⟩
⟨statement⟩        ::= ⟨variable⟩ = ⟨expression⟩
                      |⟨variable⟩ = "Error number 5"
```

```
                    |⟨variable⟩ "Error number 4"
⟨expression⟩   ::= ⟨term⟩ + ⟨expression⟩
                    |⟨term⟩ + "Error number 6"
                    |⟨term⟩ − ⟨expression⟩
                    |⟨term⟩ − "Error number 7"
                    |⟨term⟩
⟨term⟩         ::= ⟨factor⟩ * ⟨term⟩
                    |⟨factor⟩ * "Error number 8"
                    |⟨factor⟩ / ⟨term⟩
                    |⟨factor⟩ / "Error number 9"
                    |⟨factor⟩
⟨factor⟩       ::= ⟨variable⟩
                    |(⟨expression⟩)
                    |(⟨expression⟩ "Error number 11"
                    |("Error number 10"
                    |⟨number⟩
⟨number⟩       ::= ⟨digit⟩ ⟨number⟩
                    |⟨digit⟩
⟨digit⟩        ::= 0|1|2|3|4|5|6|7|8|9
⟨variable⟩     ::= A|B|C|D
```

This grammar was then used to analyse test programs. The error categories matched all the characters up to and including a semicolon and the analysis algorithm then started matching ⟨program⟩ again, i.e. a very simple error recovery.

Example of results produced:

```
 1   TEST PROGRAM;
 2   A = 3; B = A + 7;
 3   C = ((A + B) * (A − B);
                           ↑
ERROR NUMBER 11
 4   D = A + B;
 5   A = A*B* − C;
              ↑
ERROR NUMBER 8
 6   D = 4; J = 3;
            ↑
ERROR NUMBER 3
 7   A(4) = 6;
      ↑
ERROR NUMBER 4
 8   A = 3
 9   B = C;
     ↑
ERROR NUMBER 2
10   D = A
11   $
```

4. Conclusions

This paper has shown the feasibility of formally specifying compile-time error diagnostics in the grammar of a programming language, and the extent to which the positioning of these messages can be done automatically. This process is easy for a particular class of grammars (LF), but although this class of grammars is large, it appears to be impossible to write some of the more common programming languages into this form directly, although it can be done by relegating some syntactic distinctions to the semantic analysis (Appendix 1, Wood, 1969). However, the general principles outlined can probably be extended to other classes of grammars with suitable criteria chosen for the error message insertion algorithm.

References

Barron, D. W. (1971). Programming in Wonderland, *The Computer Bulletin*, Vol. 15, p. 153.

Burgess, C. J. (1971). Error diagnostics in syntax-directed compilers, Ph.D. Thesis, University of Bristol.

Foster, J. M. (1968). A syntax improving program, *The Computer Journal*, Vol. 11, p. 31.

Teitelman, W., Bobrow, B. G., Hartley, A. K., and Murphy, D. L. (1971). BBN-LISP, Tenex Reference Manual, July 1971.

Wood, D. (1969). The theory of left factored languages, *The Computer Journal*, Vol. 12, pp. 349-356; Vol. 13, pp. 55-62.

Correspondence

To the Editor
The Computer Journal

Sir

Whilst I would not support Mr. Moon's contention (this *Journal*, Vol. 15, No. 2) that Mr. Walwyn's letter (this *Journal*, Vol. 14, No. 4) should not have been published. I would support him in suggesting that Mr. Walwan's general criticisms were ill-founded.

May I suggest, sir, that you too have fallen into the trap which was so elegantly set by Professor Barron. I know a George 3/1905F installation where an ALGOL programmer can submit a program with no job control cards and no terminators. Perhaps there are installations where a customer can get the right answers with a program consisting of a single statement. What does that prove?

Surely the objective is cost-effective computing, and in this equation the cost of labour is no small proportion of the total. It is not efficient to use disc space to hold compilation 'macros' which are rarely used, or 'macros' which are so detailed that they save £1 worth of programmer's time to spend £2 extra on the machine. I have no sympathy with the incessant demand for the over-simplification of an already uncomplicated task with no regard to the cost benefit of such a policy. Perhaps Mr. Walwyn did not expect the 1907 he wished to use to be efficient.

Finally may I suggest that if a programmer appeared at my installation requiring the use of a compiler we did not have readily to hand he might very well find difficulty. Indeed if he demanded a language with which we have no experience we might very well wonder why he chose us.

Yours faithfully,
D. R. Gayler

2 Lingfield Road
Stevenage
Hertfordshire
10 July 1972

Editor's comment:

I am intrigued to learn that I have fallen into a trap, I am even more intrigued to know what that trap is.

I am also intrigued to find that to achieve the result I quoted it is *necessary* to store compilation 'macros'. Certainly the system of which I am thinking doesn't do that.

Even so, how are the figures of £1 per programmer and £2 per macro derived. One track of an EDS60 costs approx. £1. Spread over only 100 programmers this is 1p per programmer, which is 100 times more cost efficient than the programmer spending £1 worth of his time. Or have I fallen into some trap here as well?

Perhaps Professor Barron can himself throw some light on the subject. I think we should all welcome such a comment.