AUTOMATIC GENERATION OF USEFUL SYNTAX ERROR MESSAGES

by

E. Kanotoriwitz and H. Laor

Technical Report #322

June 1984
AUTOMATIC GENERATION OF USEFUL SYNTAX ERROR MESSAGES

SUMMARY

Many current compilers produce in some situations wrong error messages that mislead the user and harm his confidence in the system. It is demonstrated that a reliable and efficient syntax error handling system may be produced automatically by a compiler generator from the BNF specification of the language, and without any effort by the language implementor. This result is achieved in three ways:

(a) Some errors may not be diagnosed without knowledge of the intentions of the programmer. Some compilers employ a sophisticated analysis that attempts to capture these intentions, but which are not always successful. Such an elaborate analysis is not employed here, and instead a list of all the legal corrections is displayed, such that the programmer may readily select the right one.

(b) The recovery symbols are selected by a "careful" algorithm resulting in a high probability for correct error recovery.

(c) The "honest" error messages show also the parts of the code which could not be analyzed correctly because of errors, and where more errors may exist. Efficiency is achieved by computing the recovery sets once and for ever at compiler generation time, such that a fast error recovery at program compilation time is achieved. Experiments with erroneous programs suggest that the method compares well with the best compilers that we have seen, and is specially good in avoidance of wrong error messages.
Introduction

The requirements for good error messages are well known\textsuperscript{1,2}, but not always met. In a simple experiment\textsuperscript{3,4}, a PASCAL program with a modest error was processed by a dozen different compilers. Only one compiler produced an accurate diagnosis, while some of the others produced misleading and confusing error messages. Writing a satisfactory error message system is obviously not a trivial task. One of the difficulties is the handling of an error that may be corrected in a number of different ways. The right correction may essentially only be determined from a knowledge of the intentions of the programmer. Some compilers\textsuperscript{5,6,7,8,9,10,11} employ elaborate heuristics to make intelligent guesses of the right corrections. These methods have proved to work well in many cases but when they fail misleading messages result. As an example, consider the following error message produced by one of the most sophisticated and successful error handling systems that we have seen.

\begin{verbatim}
PROGRAM P(INPUT, OUTPUT);
PROCEDURE FACTORIAL (A);
*** SYNTAX ERROR               "PROCEDURE" expected after this token
\end{verbatim}

The correction suggested to this PASCAL program is irrelevant to the real error and the shown error location is wrong.

Error Message Principles

False error messages are both confusing and harmful to the programmers' confidence in the system. One of the prime requirements to the error handling system to be discussed in this paper was, therefore, to avoid inaccurate messages. It was our feeling that most programmers prefer simple, reliable and honest systems. The principles employed in the design of the error messages were.

1. A message that proposes how to correct an encountered error is most useful, but is only produced when there is a high degree of certainty for
its correctness, e.g.,

```java
while a > 3 do do a := a + 3;
| do SHOULD BE DELETED.
```

2. For errors that may be corrected in more than one way no attempt is made to guess which of them is the right one. Instead all the possible corrections are displayed to the programmer such that he may readily select the right one, e.g.,

```java
if a < 9 then a := b + 1;
| { a <> < > <> > } EXPECTED
```

Such a display of all the expected symbols may, in some cases, also be useful as a lesson in the capabilities of the language.

3. Pieces of code that the compiler cannot analyze correctly because of an error are underlined. The programmer will then know that unreported errors may exist in these unchecked pieces of code. This is in accordance with the requirement that the programmer may rely on the system. Consider the example:

```java
const n := 5; b := 4;
| { , , } EXPECTED
```

The first symbol that does not match with the syntax of the language is pointed at by a `|`. The symbol where the analysis is resumed (the "recovery" symbol) is marked by a `,` while the intermediate unchecked symbols are underlined.

4. The error messages reflect a simple error handling mechanism that the programmer may readily understand; i.e., when a symbol that does
not fit to the syntax is encountered the analysis stops and a list of the symbols that are expected at that position are displayed. Thereafter, the compiler skips a number of symbols until a recovery symbol is reached and the analysis is resumed. The programmer's intuitive understanding of this mechanism enables him to benefit from the error messages also in complicated situations involving errors in adjacent symbols, e.g.,

```plaintext
while b;e b-1 begin a:=a+b;

|-----| -------|<>>|    |    |

|-----| -------|<>>|    |    |
```

An Automatic and Safe Recovery Method

It was decided to develop an error recovery facility that may be produced completely automatically by a compiler generator from a BNF specification of the language, and without any effort by the language implementor. This was required not only in order to save labour, but also because we suspected that many unsatisfactory error handling systems are due to implementors who work on too tight schedules or who do not have the skills and experience required for this intricate job. A disadvantage of automatically generated error handling systems as compared to good hand written ones is that they may not be tailored to provide specially refined treatments of errors that are known to occur frequently.

Our experimental system assumes that the grammar is \( L_L^2(1) \), which facilitates the error handling \(^2\) and avoiding some of the problems discussed \(^3\). It is, however, expected that the approach is also applicable to other grammars. The paper concentrates on syntactical errors and does not discuss errors that may be detected at the later semantic analysis.

The method described in this paper may be considered as a further development of the methods of \(^4\). The basic mechanism of these methods is the
same as described in the previous section. The principal difference between our method and the previous ones is in the selection of the recovery symbols. Since we emphasize avoidance of incorrect error messages, the recovery symbols are selected more carefully. The price for using the safer recovery symbols of our system, as compared to previous systems, is that we may skip more symbols before resuming the compilation.

The following description of the error recovery method employs a graph representation of the syntax of the programming language. Each non terminal (rule) of the language is specified by a syntax diagram, e.g.

\[
\text{<if statement>}
\]

Terminal symbols are shown in circular frames while non terminals are in rectangular ones. Similar syntax diagrams have been employed for the specification of many languages, e.g., PASCAL. We have enhanced the syntax diagram notation with numbered nodes. The entry node is always number 1, while the number of the exit node is always 0. These diagrams may be considered as directed graphs. One symbol or no symbol is assigned to each arc of the graph, e.g., the terminal symbol `IF` is assigned to arc 1-2, the non terminal symbol `<statement>` is assigned to 5-0 and nothing is assigned to 5-0:

Let \( S_i \) denote the set of the terminal symbols that may follow immediately after the node \( i \), e.g.,

\[ S_i = \{ \text{THEN} \} \]

Let \( a_1, a_2, \ldots, a_n \) denote the sequence of terminal symbols that constitute the program being parsed. Let us further assume that we have parsed successfully until we reached node \( j \) of a particular syntax diagram. Thereafter, the parser
reads the terminal symbol $a_k$ ($1 \leq k \leq n$), and $a_k$ does not match with any of the symbols that is permitted after node $j$, i.e., $a_k \in S_j$. If $S_j$ has only one member, i.e., when only one arc emanates from node $j$, the encountered error is called a "first order error". If, on the other hand, $S_j$ has more than one member, i.e., when more than one arc emanate from node $j$, the error is called a "second order error". Two different error-recovery algorithms are employed for these two kinds of errors.

**First Order Error**

In this case the language permits only one particular symbol $s_j$ at the position of the encountered symbol $a_k$. This enables the error analysis to distinguish between 3 different error cases:

I. Missing symbol

II. Wrong symbol

III. Extra (superflous) symbol

Below is a PASCAL program containing these three kinds of First Degree Errors:

1 PROGRAM EXAMPLE1 (INPUT, OUTPUT);
2 VAR A, B, C: INTEGER;
3 *** | ';' expected before "INTEGER"
4 BEGIN
5 IF A>5 THEN C:=1;
6 *** | ';' expected instead of '
7 WHILE B<A, DO B:=B+1;
8 *** | '
9 END;

The algorithm which determines the error case is

```
if $a_k \in S_{j+1}$ then missing_symbol
else
  if $a_{k+1} = s_j$ then extra_symbol
  else wrong_symbol
```

where $S_{j+1}$ is the set of all the terminal symbols that are permitted immediately after the symbol $s_j$, and where $a_{k+1}$ is the terminal symbol that follows $a_k$ in the program. This Algorithm assumes that there is no error in $a_{k+1}$ (the no adjacent
error assumption). This assumption is based on the observation that the occurrence of errors in both of two adjacent symbols \(a_k\) and \(a_{k+1}\) is rare\(^{10}\). However, even in the rare cases of errors in both \(a_k\) and \(a_{k+1}\), the error messages produced by our system will show the error location correctly and the error diagnosis will, in most cases, be helpful. Here are two examples to the above case:

```
1 const a:: ;
2   if x<5 , ,x::x+10;
   l.  {end ; ; } EXPECTED
```

Second Order Error

The number of arcs emanating from node \(j\) is, in this case, greater than one. Parsing after node \(j\) may thus follow a number of different paths in the syntax diagram. In order to determine which is the right paths a number of program symbols (terminal symbols) \(a_{k+1}, a_{k+2}, \ldots, a_{k+t}\) are read until a symbol \(a_{k+t}\) that matches a "recovery symbol" is encountered. The recovery symbol is a terminal symbol from which correct parsing may be resumed with the matching program symbol \(a_{k+t}\). The symbol to be employed as recovery symbol may appear more than once in the same syntax diagram. In order to identify the instance of this symbol to be employed, the number of the syntax diagram node that precedes the recovery symbol is stored together with the symbol.

The above recovery algorithm may not work correctly if the program symbol \(a_{k+t}\) that corresponds to the recovery symbol, is also erroneous. The probability for the occurrence of an error in both \(a_k\) and \(a_{k+t}\) may, however, be expected to be low. Even if this occurs and the recovery fails, the error message will still show the error location correctly together with the list of symbols that may be expected instead of the erroneous symbol \(a_k\). The recovery algorithm assumes that there exists at least one unique recovery symbol on each of the paths that
emanates from node \( j \). These symbols constitute the recovery set of node \( j \).

Such a set will normally only have few members. The recovery algorithm, which involves a search of a recovery symbol in the \( t(t < n) \) program symbols \( a_{k+1}, \ldots, a_{k+t} \) is thus a fast process of order \( n \).

The goal of our method is to compute at the compiler generation time the recovery sets that correspond to all the nodes where a second order error is possible, i.e., for all syntax diagram nodes with more than one emanating arc. Only 54 such nodes were found in a particular grammar of PASCAL. It thus seems feasible to store all the recovery sets of a programming language together with its compiler. The method employed in our system for computation of the recovery sets from the syntactic rules of the programming language is discussed in Appendix 1, and some results of the use of these recovery sets are shown in Appendix 2.

**Experiments and Conclusions**

Appendix 2 shows the error messages produced by our system for a sample of the erroneous programs collected by\(^{19}\). The error messages are satisfactory in all the cases. It is possible to construct pathologic cases characterized by clusters of adjacent errors, where our error recovery method may fail. A statistical analysis of errors in 600 student programs suggests, however, that the probability for a number of adjacent errors is low. Furthermore, even in these rare cases where the error recovery fails, the error message will show the error location correctly together with a useful error diagnosis.

The general impression from the experiments is that the error recovery and error messages produced by our system compares well with the best systems that we have seen. Our system is particularly good in avoiding of misleading messages. This is partly due to the careful handling of second order errors, i.e., errors that may be corrected in more than one way. We do not suggest a single correction to such an error on the basis of an intelligent guess of the programmer's intentions, but display instead all the possible corrections such that the programmer may
readily select the right one. Furthermore, the parsing is resumed at recovery symbols that are selected carefully such that there is a high degree of certainty that the analysis may continue correctly. The algorithm for selection of these recovery symbols is one of the contributions of this paper.

An interesting aspect of the method is that the selection of the recovery sets may be done completely automatically by a compiler generator from the specification of the syntax of the language. The language implementor may thus obtain a satisfactory error handling system without any effort. The grammar of PASCAL has only 54 nodes where second order errors may occur, i.e., only 54 sets of recovery symbols have to be stored in order to achieve an extremely fast error recovery.

Acknowledgement:

The authors wish to thank G. David Ripley for the sample of PASCAL syntax error data which were employed in the testing of our system.
REFERENCES


[10] Pen nell o, T.J. and De Remer, F., A forward move algorithm for LR error recovery. In: Conference record of the 5th annual ACM


APPENDIX 1 - A METHOD FOR COMPUTING THE RECOVERY SETS

At time of compiler generation all the syntax diagrams (rules) of the language are searched for nodes where a second order error is possible, i.e., nodes from which more than one arc emanate. For each of these nodes a recovery set is computed.

Let \( j \) denote the number of such a node. An example is found in the following diagram of a PASCAL <case statement>, where \( j = 4 \).

![Diagram of PASCAL case statement]

It is required to find at least one unique recovery symbol for each of the paths that emanates from node \( j \). In our example there are two arcs 4-5 and 4-9 that emanate from the node \( j = 4 \), and the number of emanating paths is thus at least 2. The members of the recovery set, which is denoted \( \text{REC_SYM} \), belong to three different categories:

I. Terminal symbols that appear in the actual syntax diagram on the paths emanating from node \( j \). These symbols are assigned a symbol type \( \text{SYM_TYPE} \) called TERMINAL, e.g.,

\[
\text{SYM_TYPE(':'')} = \text{TERMINAL}
\]

II. The terminal symbols that may head the non terminals that appear in the actual syntax diagram in the paths emanating from node \( j \). These symbols are assigned the symbol type HEAD, e.g.,

\[
\text{SYM_TYPE('+'')} = \text{HEAD}
\]

In our example two such non-terminal appear i.e., <constant> and <state-
Consider for instance:

The terminal symbol that may head `<constant>` are `'+', '-', string, unsigned_integer, unsigned_real and constant_identifier. A member of the head of a non terminal may, in some cases, not be applicable as a recovery symbol, for instance because the same symbol is already a recovery symbol in another path. In this case, a recovery symbol is searched beyond the non terminal. The "internal symbols" of the non terminal, i.e. all the terminal that may appear in the non terminal after the head symbols, are not permitted as recovery symbols. These terminal symbols constitute a set to be called NON_REC_SYM. The non terminal `<constant>` of the above example contributes three terminal symbols to the NON_REC_SYM set: unsigned_integer, unsigned_real, constant_identifier. These symbols are in this case also in the head of `<constant>`.

If no recovery symbol is found inside the actual syntax diagram in some of the paths emanating from node $j$, a boolean (flag) WITH_FOLLOW is set to TRUE. It may, in this case, be necessary to recover with one of the terminal symbols that can follow immediately after the actual syntax diagram. These terminal symbols constitute a set to be called FOLLOW. This set may be computed by analysis of all the syntax diagrams that contain the actual syntax diagram.
In the experimental system FOLLOW is computed at the time of compilation, when a second order error is encountered. The advantage of this approach is that FOLLOW may be computed from actual content of the stack of the parser resulting in a more precise and reliable error recovery. Consider for instance the situation where the `<case statement>` of the above example is embedded in an `<if statement>`:

\[
\text{FOLLOW}=\{; '\text{END}', '\text{ELSE}'\}
\]

If on the other hand, the `<case statement>` is not embedded in an `<if statement>`, FOLLOW is reduced to:

\[
\text{FOLLOW}=\{; '\text{END}'\}
\]

In the experimental system the REC_SYM and NON_REC_SYM sets as well as WITH_FOLLOW are computed at the compiler generation time and stored with the compiler. The recovery algorithm is:

\[
\text{repeat}
\]

\[
\text{read next input symbol } a_v,
\]

\[
\text{if } a_v \in \text{REC_SYM then } \text{DONE:=true;}
\]

\[
\text{else if WITH_FOLLOW then }
\]

\[
\text{if } a_v \in (\text{FOLLOW}\setminus\text{NON_REC_SYM})
\]

\[
\text{then } \text{DONE:=true;}
\]

\[
\text{until DONE;}
\]

Recover with \(a_v\) at its corresponding node;

The computation of REC_SYM and NON_REC_SYM sets starts with an initialization phase:

a) Compute the set \(S_j\) of terminal symbols that may follow immediately after node \(j\). In our example, a search of safe recovery symbols will be started from:

\[
S_4=\{';'+', '-', '\text{string}', '\text{END}', \text{unsigned_integer}, \text{unsigned_real}, \text{constant_identifier}\}\]

b) The members of \(S_j\) are useful as recovery symbols either when the error at node \(j\) is of the "extra symbol" type (three error types are defined in the section
on first degree errors), or when there is an arc that brings us back to node \( j \).

A search of safe recovery symbols will be started from a set of nodes that is
called NODES. This set comprises of one node on each one of the arcs that
emanate from node \( j \). These nodes are computed as the FOLLOW-NODEs of the
terminal symbols in \( S_j \), i.e.,

\[
\text{NODES} = \{ \text{FOLLOW-NODE}(s) \mid s \in S_j \}.
\]

The FOLLOW-NODE of a terminal symbol \( s \) is computed as:

\[
\text{FOLLOW-NODE}(s) =
\begin{cases} 
\text{node immediately after } s & \text{when } \text{SYM\_TYPE}(s) = \text{TERMINAL} \\
\text{node immediately after the non terminal that is headed by } s & \text{when } \text{SYM\_TYPE}(s) = \text{HEAD}
\end{cases}
\]

The results of all the above calculations are in our example:

\[
\text{NODES} = \{6, 0\}
\]

b. Initialize the set NON-REC_SYM of the terminal symbols that are not permitted
as recovery symbols (see point II above). This set is initialized as the union of the
contributions of all the non terminals that may appear immediately after node \( j \).
The terminal symbols that may appear in these non terminals after the head sym-
\bols are entered NON-REC_SYM. In our example we initialize:

\[
\text{NON-REC_SYM} = \{\text{unsigned\_integer, unsigned\_real, constant\_identifier}\}
\]

d. Initialize the set of recovery symbols to be

\[
\text{REC_SYM} = S_j \setminus \text{NON-REC_SYM}
\]

In our example we initialize:

\[
\text{REC_SYM} = \{\text{'+', '-', string, 'END'}\}
\]

e. Initialize WITH\_FOLLOW = FALSE.

It is now required to find a recovery symbol on everyone of the paths that
emanates from node \( j \). The search on these paths starts from the nodes that are
members of NODES (it contains at least one node for each of these paths). This is accomplished with a "breadth search" over all the paths. The algorithm is given below in a PASCAL like program, where (* *), brackets, are employed to embrace comments:

```pascal
DONE := \{\};
for each node i in NODES while NODES is not empty do
begin
NODES := NODES - \{i\};
DONE := DONE + \{i\};
if i = 0 (* the exit node is encountered *) then
WITH FOLLOW := true;
else
for each s_i in S_i do
(* S_i = \{s_i | s_i is a terminal that may follow immediately after node i \} *)
begin
if (s_i \in NON_RECURSYM) and (FOLLOW_NODE(s_i) \notin DONE) then
NODES := NODES + FOLLOW_NODE(s_i);
else
if (s_i \notin NON_RECURSYM) and (s_i \notin REC_SYMS) then
REC_SYMS := REC_SYMS + \{s_i\};
else (* Let s_iold denote the instance of s_i that is already in REC_SYMS, and let us assume that the FOLLOW_NODE and SYM_TYPE of this old instance have been stored *)
begin
if s_i and s_iold are the same instance then
leave s_iold in REC_SYMS
else (* both s_i and s_iold do not determine a unique recovery, and can therefore not be employed for recovery *)
begin
DENY_RECOVERY_OF(s_i)
DENY_RECOVERY_OF(s_iold)
end;
end;
end;
```
procedure DENY_RECOVERY_OF(s_k):
begin
    REC_SYM:=REC_SYM-{s_k}
    NON_REC_SYM:=NON_REC_SYM+{s_k}
    if FOLLOW_NODE(s_k) & DONE then
        NODES:=NODES+FOLLOW_NODE(s_k)
    if SYM_TYPE(s_k) = HEAD then
        (* let POST(s_k) denote the set of all the
         terminals in the non terminal headed by s_k,
         and which may appear after s_k. *)
        for each s_f in POST(s_k) do
            if s_f in REC_SYM then
                DENY_RECOVERY_OF(s_f)
            else
                NON_REC_SYM:=NON_REC_SYM+s_f
        end;
end;

At the end of the process we obtain:

REC_SYM={'+', '-', 'string', 'END', ':', ';'}
NON_REC_SYM={"unsigned_integer, unsigned_real, constant_identifier"}" WITH_FOLLOW=true;

It is possible to construct a grammar such that some paths will not have a
recovery symbol. This may happen when some of the symbols of FOLLOW must be
deleted because they belong also to the NON_REC_SYM set. It is, however, our
feeling that such situations are rare in practical languages. The compiler genera-
tor may be designed to warn the language implementor of paths with no recovery
symbols. The implementor may then remedy the grammar of the language from
these weaknesses. It is remarked that even if this is not done, and the recovery
fails because of lack of a recovery symbol on some path, the error message at
node j is still useful.
APPENDIX 2 - EXAMPLES OF ERROR RECOVERY AND ERROR MESSAGES

begin
  while a>9 do x:=x+1
  if a<9 then v.+v+1;

begin
  if x:=0 then x:=1;

begin
  if x:=0 then
    f:=f;

procedure pr;
  begin
  end

procedure gen;
  begin
    var q:
    begin
    end;

begin
  begin
    x:=x+1;
  end;

begin
  begin
    {end ; } EXPECTED
  end.

begin
  if a<9 then v.+v+1;

begin
  if x:=0 then x:=1;

begin
  if x:=0 then
    f:=f;

procedure pr;
  begin
  end

procedure gen;
  begin
    var q:
    begin
    end;

begin
  begin
    x:=x+1;
  end;

begin
  begin
    {end ; } EXPECTED
  end.

begin
  if a<9 then v.+v+1;

begin
  if x:=0 then x:=1;

begin
  if x:=0 then
    f:=f;

procedure pr;
  begin
  end

procedure gen;
  begin
    var q:
    begin
    end;

begin
  begin
    x:=x+1;
  end;

begin
  begin
    {end ; } EXPECTED
  end.

begin
  if a<9 then v.+v+1;

begin
  if x:=0 then x:=1;

begin
  if x:=0 then
    f:=f;

procedure pr;
  begin
  end

procedure gen;
  begin
    var q:
    begin
    end;

begin
  begin
    x:=x+1;
  end;

begin
  begin
    {end ; } EXPECTED
  end.
1.9

var a,x,m,n;

begin
end.

begin
a:=4, b:=5, c:=7, s:=9, v:=10;
end.

begin
x:=3*(5+6);
end.

begin
x:=4*x+5;
end.

const n:=5, b:=4;

var a,b,c,d,e;

begin a:=5*b, c:=o-d;
if a>9 than a:=b+1;

while b:=b+1 begin a:=4+b;

if a>3 than a:=a+1;

while a>3 r do a:=a+3;

end;
begin
a:=b+;
if a>b*(a+c) then d:=d-2;
end;
begin
a:=5*b+c:=d a:=9, b:=8;
end;
end;
end;
if a=b then a:=a+b;
c:=c*(a+b);

while d>5 do call ada;
if d< then a:=a+b;
end.