SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 14(11), 1095–1107 (NOVEMBER 1984)

A Language Independent Error Recovery Method for LL(1) Parsers

MICHAEL SPENKE, HEINZ MÜHLENBEIN, MONIKA MEVENKAMP, FRIEDEMANN MATTERN* AND CHRISTIAN BEILKEN*

Gesellschaft für Mathematik und Datenverarbeitung, POB 1240, D-5204 St. Augustin, Germany

SUMMARY

An efficient and systematic LL(1) error recovery method is presented that has been implemented for an LL(1) parser generator. Error messages which provide good diagnostic information are generated automatically. Error correction is done by discarding some input symbols and popping up some symbols from the parsing-stack in order to restore the parser to a valid configuration. Thus, symbol deletions and insertions are simulated. The choice between different possible corrections is made by comparing the cost of the inserted (popped) symbols with the reliability value of the recovery symbol (the first input symbol that is not discarded). Our concept of reliability is based on the observation that input symbols differ from each other in their ability to serve as recovery points. A high reliability value of a symbol asserts that it was probably not placed in the input by accident. So it is reasonable not to discard that symbol but to resume parsing. This is done even if a string with high insert-cost has to be inserted before that symbol in order to fit it to the part of the program that has already been analysed. The error recovery routine is invoked only when an error is detected. Thus, there is no additional time required for parsing correct programs. Error-correcting parsers for different languages, including Pascal, have been generated. Some experimental results are summarized.

KEY WORDS Syntax error recovery LL(1)-parsing Parser-generator Backtrack-free correction

INTRODUCTION

Error recovery is one major task of compilers for high-level programming languages. The handling of *syntactic errors* is important, because modern compilers are syntax-directed, and thus the syntactic analysis of erroneous programs forms the basis for the detection of other types of errors. A complete and precise error diagnosis is of *great practical importance*: the detection of all errors in one compiler pass with no spurious errors may lead to a considerable speed-up of program development.

The interest in systematic and language independent error recovery strategies has increased considerably in the past ten years. The main reason for this development is the automation process in compiler-design itself (parser generators, compiler-compilers). An automatically generated parser into which error recovery is built later by hand seems to be of little value!

Dissertations by Leinius, LaFrance, Levy and Peterson dating from the early 70s are milestones in this development. They investigate for the first time the problems of systematic syntax error recovery based on context-free grammars.

*Present address: Souderforschungsbereich 124, Universität Kaiserslautern POB 3049, D-6750 Kaiserslautern, Germany.

0038–0644/84/111095–13$01.30
Received 18 February 1983

The situation before the 70s is best illustrated by two quotations. Pollack[1] writes: Chapter 5 treats the most poorly documented topics in compiler writing, those of error detection and error correction. Few general techniques are currently known for these subjects, perhaps accounting for the dearth of available literature.' Feldman and Gries[2] write: 'There has been very little effort on the problem of automatic error detection and recovery in syntax-directed processors. Once again, even a bad system would be of great value to the users!'

In the 70s a variety of error recovery methods were developed. Best known is the strategy developed by Wirth and Amman[3] which is used in most Pascal compilers. However, it is systematic to a certain extent only, and it contains a number of language-dependent heuristics. The suffix analysis developed by Moll[4] is not yet operable in practice and does not permit checks of static semantics after the first error. Fischer, Milton and Quiring[5] describe an experimental method which corrects all errors by mere symbol insertions. Parts of the method developed by Pai and Kieburtz[6] are similar to the method described here. There is also a number of methods which were designed for LR- and precedence-parsers. Graham and Rhodes[7] and Graham, Haley and Joy[8] are mentioned most frequently. Another interesting method has been developed by Lewi *et al.*[9] It can be used for both top-down and bottom-up parsers. A good survey of various methods is presented by Röhrich.[10]

Ripley and Druseikis[11] have made a statistical analysis of the errors committed by programmers and have set up an error benchmark. They criticize that hardly any of the suggested methods has been used in practice. 'Most evaluations in print consist at best of an illustration of the technique on a few examples, almost invariably leaving the reader wondering how representative these examples are, and how well the technique works in general.' Aho and Ullman[12] confirm that an investigation of this topic is still valuable: 'Although a considerable amount of theoretical and practical effort has been expended in exploring recovery and repair techniques for syntactic errors, the optimal strategy for any programming language is still an open question.'

A method to be used in a compiler generator should have the following properties:

(a) language-independent
(b) efficient
(c) automatic generation of user-oriented error messages (no language-dependent messages)
(d) suited for compilers, not only for parsers (analysis of static semantics enabled, no changes of the grammar required)
(e) detection of all errors, no spurious errors.

It should be mentioned that it cannot be exactly decided if the last point is met, since 'spurious error' and 'error position' are pragmatical terms which cannot be formalized completely.

The usual analysis methods (LL, LR) have the *valid-prefix property*, i.e. they detect an error as soon as the read input is no longer any prefix of a syntactically correct program. The following example shows that the error position defined by the valid-prefix property is not always identical with the 'real' error position:

```
VALUE := 0.3*(A+B)+0.7*C+D);
                          ↑
```

It cannot be decided if and where an opening bracket is missing or whether a closing bracket is redundant. A parser cannot figure out the 'real' error. Therefore it should confine itself to 'regaining foothold' as soon as possible and to continue program analysis, i.e. to carry out a *quick and secure recovery*. A correction in the strict sense can be in principle carried out only by the programmer.

In the LL(1)-case further restrictions usually have to be met by an efficient error-correcting parser:

1. The input is read from left to right in one single pass.
2. Just as in the normal analysis, in case of an error the look-ahead is limited to one symbol.
3. The input already accepted cannot be corrected.
4. No backtracking, even in the case of an error.

According to these restrictions, in the following example the parser has to decide immediately whether the semicolon terminates the if-statement or whether it is misplaced and then will come later.

```
if X = 5; ////////////
```

When the next input symbol is read a wrong decision cannot be annulled.

THE METHOD

The error-correcting parser presented here is based on the table-driven (strong) LL(1)-analysis, as described by Aho and Ullman.[12] The parsing method is slightly extended to ensure an even earlier error detection. If an error occurs, the error recovery routine (ERR) is called. The ERR performs a *correction*, so that the LL(1)-analysis can be continued. A correction is a modification of the parser configuration by skipping some input symbols ($T_e . . T_{r-1}$) and popping up some symbols ($A_1 . . A_{i-1}$) from the stack, which leads to a valid configuration (see Figure 1).

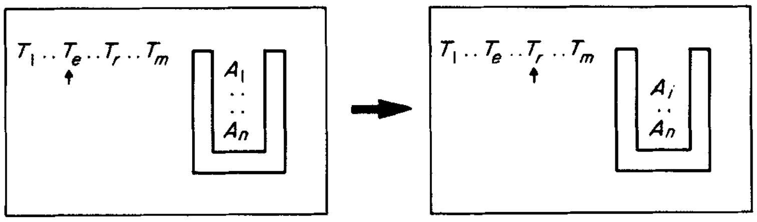

Figure 1. Parser reconfiguration

The strong LL(1)-parser detects an error as soon as the current input symbol T_e is not contained in $FIRST(A_1.FOLLOW(A_1))$. Therefore an error is detected if T_e cannot be derived at the first position from A_1 (or equals A_1). In the case that the empty word *LAMBDA* can be derived from A_1, an error is only detected if T_e can never follow A_1 in any sentential form. However it is certain that there is an error unless $T_e \in FIRST(A_1 . . A_n)$. If *LAMBDA* can be derived from A_1, a valid configuration is not yet guaranteed if T_e is only a symbol which may in principle follow A_1 (is in $FOLLOW(A_1)$). Instead it is necessary that T_e can follow A_1 in this specific situation, i.e. $T_e \in FIRST(A_2 . . A_n)$. Therefore, the strong LL(1)-parser in some cases performs moves, though it has already encountered an erroneous symbol.

By an additional inspection of the stack (for computing $FIRST(A_2 . . A_n)$) before applying *LAMBDA* rules (popping non-terminals from the stack) the *immediate-error-detection property* (IEDP) is obtained. The IEDP has been proposed by Fischer, Tai and Milton.[13] This property is even stronger than the valid-prefix property mentioned above: a parser with the IEDP never performs a move when it has encountered the erroneous symbol. Fischer, Tai and Milton[13] further show how to obtain the IEDP without additional effort for parsing correct program passages. (In grammars where *LAMBDA* can be derived indirectly from a non-terminal A, the stack has to be inspected in some additional cases as well.) Ghezzi[14] shows how to obtain the IEDP without parser modification, but by extending the grammar.

Analysing the erroneous input if P**2/4;-Q>0 then . . a parser without the IEDP will still perform the derivation $\langle \textit{expr-rest} \rangle \rightarrow LAMBDA$, whereas a parser having the IEDP will immediately stop when the semicolon is detected, thus maintaining the important information that the expression will possibly continue.

In a parser with the IEDP an important prerequisite for error recovery holds: in case of an error all terminal suffices for the input already accepted can be derived from the stack: $A_1 . . A_n \overset{*}{\Rightarrow} w$ if only and only if $T_1 . . T_{e-1} w \in L(G)$. Since T_e is erroneous, $A_1 . . A_n \overset{*}{\Rightarrow} T_e . . T_m$ does not hold.

If we assume that any incorrect program is a 'near miss', i.e. that it is strongly similar to a correct program, there exists an r $(e \leqslant r \leqslant m)$ and an i $(1 \leqslant i \leqslant n)$ such that the following holds: $A_i . . A_n \overset{*}{\Rightarrow} T_r . . T_m$ (unless the program includes further errors). By skipping $T_e . . T_{r-1}$ and popping $A_1 . . A_{i-1}$ the parser can be reconfigured. It is easy to see that popping the symbols $A_1 . . A_{i-1}$ from the stack corresponds to the virtual insertion of the symbols at the error position. Thus, the ERR does not correct the incorrect program in the sense that it transforms it into a correct terminal string. However, each incorrect program is transformed into a *sentential form*, namely $T_1 . . T_{e-1}\ A_1 . . A_{i-1}\ T_r . . T_m$. In this way, a whole class of possible corrections is suggested by the ERR. The resulting error messages are better understandable to the programmer than a message reporting the insertion of some specific terminal string that can be derived from $A_1 . . A_{i-1}$.

The method described performs only those insertions which can be simulated by popping the stack. Consequently, only those symbols are used as recovery points which will certainly become legal followers in a future configuration regardless of the actual recovery action that will eventually be used. Thus, the ERR uses a dynamically computed set of symbols which are already 'expected' in the sense described above. The set of recovery points can even be increased by expanding those non-terminals for which there is a unique rule, immediately after pushing them into the stack. For example, when $\langle \textit{compound\_stmt} \rangle$ is pushed into the stack it can immediately be replaced by begin$\langle \textit{stmt_list} \rangle$ end. As a consequence all statement beginnings and end are turned into potential recovery points.

Of course, the ERR cannot foresee for given r and i whether $A_i . . A_n \overset{*}{\Rightarrow} T_r . . T_m$ will be true, since it knows the input until T_r only. Therefore, it can only guarantee that $T_r \in FIRST(A_i)$, and thus a valid configuration is obtained.

Since in general there is more than one correction possible, it is the task of the ERR to select a *plausible* correction: According to certain *heuristics* a correction is selected for which $A_i . . A_n \overset{*}{\Rightarrow} T_r . . T_m$ is most likely and for which only a short part of the program is skipped.

The heuristics are based on certain *assumptions* concerning the errors committed by programmers:

1. An incorrect program is usually very similar to a correct program.
2. An error is very soon followed by a correct piece of a program.
3. There are important symbols (such as keywords) which are omitted quite rarely, whereas other, less important, symbols (such as the redundant characters comma or semicolon) are frequently omitted.
4. There are very reliable symbols which most likely do not occur in the input by accident or error and which always occur in a specific context (e.g. a then is always part of an if-statement).
5. If an error cannot be corrected by deletion and/or insertion of a few symbols, the reason is often a complete, misplaced syntactic unit, such as a whole expression where only a single constant is allowed or a type-declaration following the variable-declarations (which is not allowed in Pascal).
6. A frequent reason for syntax errors is typing errors. In addition, similar basic symbols, such as round and square brackets or ':' and '. .' may easily be confused.

The first two assumptions are taken into account by first looking for such corrections where only few symbols are skipped: the symbols following the error are examined in sequence to determine whether they can serve as a recovery point.

To consider the third point, insert-costs are assigned to any terminal symbol of the language, as proposed by Fischer, Milton and Quiring.[5] The function *INSERT-COST* can be extended to strings of terminals:

$$INSERT\text{-}COST(aw) = INSERT\text{-}COST(a) + INSERT\text{-}COST(w)$$

The *INSERT-COST* of the cheapest string derivable from X is assigned to a non-terminal X ($INSERT\text{-}COST(LAMBDA) = 0$). In a straightforward way the *INSERT-COST* function can be extended to strings of terminals and non-terminals.

Whereas Fischer, Milton and Quiring[5] consider all insertions (and only these), our method considers only those insertions which can be simulated by popping symbols from the stack. Consequently, only those symbols are inserted which are definitely expected, but which have not occurred. On the one hand this reduces the danger of an erroneous insertion and on the other it simplifies the method very much. The *INSERT-COST*s are not necessary for choosing among various insertions in front of the same recovery point; instead, they are used to decide whether an input-symbol should be used at all for recovery or whether it should be skipped. If an important, reliable symbol is found in the input, the ERR shall be allowed to insert (assume as missing) even an expensive string. A less reliable symbol, however, should be used for recovery only if it fits very well with the input read so far and if no expensive insertions are necessary. Therefore, in addition to the *INSERT-COST* the *RELIABILITY* of a terminal-symbol is used here. The *RELIABILITY* indicates how a symbol is suited as a recovery point. A symbol of low reliability is more easily skipped by the ERR, whereas a symbol of high reliability is used as a recovery point even if the insertion of a whole sequence of symbols is necessary. The end-of-file symbol always has infinite *RELIABILITY*, since the ERR can never ignore it.

The above considerations lead to the following definition:

Definition

A correction by deletion of $T_e. .T_{r-1}$ and insertion of $A_1. .A_{i-1}$ is called *plausible* $\Leftrightarrow INSERT\text{-}COST(A_1. .A_{i-1}) < RELIABILITY(T_r)$.

The error recovery routine is now easily explained: the ERR selects that plausible correction in which the fewest input-symbols are skipped.

Some examples will now be given to illustrate the method and provide insight into the scheme for specifying the two tables for Pascal.

```
A := (B*C; ////////     A := (B*C+; /////////     A := (B*(C+; /////////
```

In the first case it is clearly plausible to assume that a closing bracket is missing before the semicolon. Therefore the following should be true:

$$INSERT\text{-}COST(')') < RELIABILITY(';')$$

In the second case $\langle TERM\rangle$ and ')' have to be inserted if the semicolon is used as a recovery point. Since the cheapest terminal string derivable from $\langle TERM\rangle$ should be an identifier, it is examined whether

$$INSERT\text{-}COST(\langle TERM\rangle')') = INSERT\text{-}COST(ID\ ')')$$
$$= INSERT\text{-}COST(ID) + INSERT\text{-}COST(')') < RELIABILITY(';').$$

In the second case it is certainly reasonable to use the semicolon as a recovery point. In the third case however it is quite questionable what the ERR should do. According to the definition of the two tables, the semicolon is used as a recovery point here (which may lead to spurious errors if the expression continues) or the semicolon is skipped and a following, more reliable symbol (e.g. the next end) is used as a recovery point (which may lead to undetected errors in the skipped statements before the end).

By the definition of the two tables different strategies of error recovery can be implemented. The following special cases ought to be mentioned:

1. If all *RELIABILITY* values are set to zero, everything following an error is skipped until the EOF symbol (which has infinite *RELIABILITY*). This will never result in spurious errors, but only the first error in the program will be detected.
2. If all *INSERT-COST*s are set to zero and all *RELIABILITY* values are set to infinity the first possible correction will always be found to be plausible and therefore carried out. Very little is skipped and only few errors can be overlooked in this case, but the danger of spurious errors is great, since even extremely implausible corrections are selected, as illustrated in the following Pascal example:

   ```
   ... begin ... begin A := . ////////////
   ```

   The ERR would assume that the dot terminates the whole program and that $\langle expr\rangle$ end end were missing.
3. If the *RELIABILITY* and *INSERT-COST* values of all terminal symbols are set to 1, the ERR will carry out a correction only if it can be done without the insertion of any symbol. The result is a deletion-only method, i.e. all errors are recovered by mere skipping to the next legal follower.
4. If almost all *RELIABILITY* values are set to zero and some few symbols are provided with infinite *RELIABILITY*, the panic-mode is obtained: a statically fixed set of stop symbols is defined, which can serve as recovery points.

Apparently the definition of the two tables induces a trade-off between quick recovery and secure recovery; in other words, between few errors overlooked and few spurious errors.

The kind of considerations, which have lead to the specification of the two tables for Pascal, can be illustrated by another example:

```
if X = 5 ; ////////////        if X = 5 else ////////////
```

In the first case it should rather be assumed that the semicolon was misplaced than that the then was missing. Therefore the following should be true: *INSERT-COST*('then') > *RELIABILITY*(';'). In the second, almost equivalent case, however, it is reasonable to assume that the then-part has been omitted.

The following *recommendations* for defining the tables can be derived from the examples discussed here and from a number of practical tests:

(a) The *INSERT-COST*s of special characters, numeric constants and identifiers are between 1 and 5.

(b) Keywords have *INSERT-COST*s between 20 and 30 depending on their length.

(c) The *RELIABILITY* of a symbol is usually set to three times its *INSERT-COST*. Thus it is, for example, possible to insert two other keywords as well as some other symbols in front of a keyword found in the input.

(d) Identifiers are provided with the *RELIABILITY* 1 since identifiers occur very frequently and in many different contexts. Furthermore, an identifier is often obtained by a typing error in a keyword.

(e) Some symbols are defined as stop symbols and have an extremely high reliability. Thus it can, for example, be ensured that an end is always matched to the last begin. In this way it is possible to recover even after wrong ERR assumptions.

FURTHER ALGORITHMS FOR ERROR RECOVERY

In order to take into account also the fifth and sixth assumptions on the errors commited by programmers the ERR is extended in two additional ways:
the *phrase-parser* and the *spelling correction.*

The phrase-parser avoids the skipping of longer input passages without any analysis and enables the analysis of misplaced syntactic units, such as declarations in the statement part: if a reliable symbol is found in the input which cannot serve as a recovery point (since it is not expected in the stack), it is checked whether the symbol signals the beginning of a unique syntactic phrase. In this case the phrase is pushed onto the stack and the parser is called recursively. Thus the isolated unit is analysed and skipped as a whole. Afterwards the search for a recovery point is resumed.

If, for example, a begin is encountered by the ERR, a whole ⟨*compound_stmt*⟩ is first analysed and then the ERR is resumed. Symbols within the ⟨*compound_stmt*⟩ can therefore not be used as recovery points.

A case that occurs quite frequently in Pascal, namely that the order of declarations is confused, can be handled by the phrase-parser: if, for example, a type-declaration follows the variable-declarations, type is the erroneous symbol and it cannot be used as a recovery point. Since type starts a unique phrase, the phrase parser can analyse a type declaration and subsequently the ERR can, for example, use the keyword procedure as a recovery point.

In particular, the phrase-parser can also analyse statements following the syntactic end of the program, though the stack is already empty when they are encountered.

Starting points for the phrase-parser may be those symbols which are of high reliability and which have a unique right context. For example, in Pascal then is always part of the phrase then ⟨*stmt*⟩ ⟨*else-part*⟩. Also do has a unique right context (do ⟨*stmt*⟩) though it may occur in two different statements.

Another important reason for syntax errors is *typing errors*. For correcting typing errors the erroneous symbol is compared with the set of symbols expected instead. Because of the immediate-error-detection property (see above) $FIRST(A_1 . . A_n)$ is exactly the set of legal followers at the error position. It is checked whether the erroneous symbol is similar to one of the legal followers.

Two strings of at least three characters are defined to be *similar* if one of the following conditions holds:

(i) They have the same length and differ in one character only.
(ii) One string can be obtained by adding one character to the other at an arbitrary position.
(iii) One string is produced from the other by interchanging two neighbouring characters.
(iv) One string is a prefix of the other one.

Furthermore, certain non-standard characters, such as round and square brackets, ':' and '. .' are defined to be similar. This definition covers about 80 per cent of all typing errors.[15]

If T_e is similar to a symbol $X \in FIRST(A_1 . . A_n)$, T_e is corrected to X. If X is a prefix of T_e, T_e is divided into two symbols.

Spelling correction is complicated by the fact that a typing error in a keyword usually produces an identifier. Frequently the syntax also allows an identifier instead of the keyword, so that the error cannot be detected at once. In A := B; whyle A = C do . . the parser has to assume when encountering whyle that an assignment will follow. The error will be detected only when encountering A. This would be too late for spelling correction.

This problem is solved as follows. The scanner examines whether a declared or an undeclared identifier has been encountered. The syntax requires that an assignment begins with a declared identifier. In this way the error is already detected one symbol earlier and can be corrected by spelling correction.

If an undeclared identifier is encountered and a declared identifier is expected, the symbol-table is examined for an identifier which is similar to the erroneous symbol and which has already been declared.

THE ERROR RECOVERY ROUTINE

The ERR actions can be described as follows: the ERR looks for the smallest r $(e \leqslant r \leqslant m)$ for which there is an i $(1 \leqslant i \leqslant n)$, so that the following is true:

(a) $T_r \in FIRST(A_i)$
(b) $INSERT\text{-}COST(A_1 . . A_{i-1}) < RELIABILITY(T_r)$

Note: (1) $A_1 . . A_{i-1}$ denotes for $i = 1$ the empty string *LAMBDA*. (2) Since T_m = EOF and A_n = EOF and since the *RELIABILITY* of EOF is infinite, there is always such an r.

For the obtained r the smallest valid i is selected and the following recovery action is performed:

(i) $T_e. .T_{r-1}$ are skipped (deletion).
(ii) $A_1. .A_{i-1}$ are popped from the stack (insertion).
The following algorithm also incorporates the phrase-parser and spelling correction.

Algorithm for parser reconfiguration

A1 ... An	stack contents. STACKTOP = A1
TOKEN	the current input symbol
UNIQUE(TOKEN)	states whether TOKEN starts a unique phrase
UNIQUE_PHRASE(TOKEN)	indicates the unique phrase starting with TOKEN
PREDICT(PHRASE)	writes a phrase on top of the stack

```
procedure ERROR_RECOVERY;
begin
    loop for X in FIRST(A1. .An) do
        if SIMILAR(TOKEN,X)
          then begin TOKEN: = X; (* spelling-correction*)
                          return;
                     end;
    endloop;

    if TOKEN = <NEW_IDENTIFIER> and <OLD_IDENTIFIER> in FIRST(A1. .An)
      then (* misspelt or undeclared identifier *);

    loop until RECOVERY do

        loop for I in 1. .N
             while INSERT_COST(A1...Ai-1) < RELIABILITY(TOKEN)
          do
             if TOKEN = Ai or TOKEN in FIRST(Ai) then signal RECOVERY;
        endloop;
        if UNIQUE(TOKEN)
          then begin (* phrase-parser *)
                      PREDICT(UNIQUE_PHRASE (TOKEN)); (* push*)
                      (* recursive call of the parser *)
               end
          else GET_NEXT_TOKEN; (* deletion *)
        exit RECOVERY is
          POP(A1...Ai-1); (* insertion *)
    endloop;
end;
```

First it is checked whether the error can be corrected by spelling correction. If this is impossible, the until-loop is executed until the event RECOVERY is signalled and the exit block is executed.

Apart from the grammar the two tables with *INSERT-COST* and *RELIABILITY* values present the only information about the specific language the ERR knows. It does not have any language-dependent heuristics, such as 'In Pascal the order of declarations is frequently confused'.

The proposed algorithm has to meet the restrictive LL(1)-prerequisites and is easy to implement. Nevertheless, when compared in practice with other, more sophisticated methods tailored specifically to Pascal, it yielded at least equivalent results (see below).

AN EXAMPLE

The proposed method was implemented in an LL(1)-parser generator, and an error-correcting Pascal parser was generated. The *INSERT-COST* and the *RELIABILITY* were specified for each terminal-symbol of Pascal. The Pascal grammar used is however not unambiguous because of the well-known dangling-else problem and therefore it does not possess the LL(1)-property. Though unambiguous grammers can be constructed for ⟨*if-statement*⟩, according to Aho and Ullman[12] there is no such LL(1)-grammar. The parser generator signals the violation of the LL(1) condition, but generates an analysis matrix matching each else to the last if.

The example in Figure 2 shows how the Pascal parser reacts to a *demonstration program*. The error lines are marked with arrows, the erroneous symbol is underlined. Skipped passages are underlined by a broken line. The error messages indicate how the ERR has corrected the program so that the programmer can easily see whether the assumptions of the ERR are correct or whether he has to expect spurious errors. Note that all error messages are generated automatically and are not tailored specifically to Pascal.

Explanatory notes concerning error messages

Line 14: Because of the IEDP the ERR knows all legal followers. One of them (else) is similar to the erroneous symbol, therefore spelling correction.

Line 15: The scanner examines if and where an identifier has been declared and returns a corresponding token: ⟨*new-identifier*⟩, ⟨*local-identifier*⟩ or ⟨*global-identifier*⟩. The error is thus detected earlier and spelling correction is still possible.

Line 16: Spelling correction by comparison with the set of all declared identifiers.

Line 17: If a spelling correction is impossible, a declaration error is assumed.

Lines 14–17: Correction at token level.

Line 18: Recovery by skipping (deletion).

Line 19: CAR = DAMAGED is already a complete expression. Recovery by or is only possible thanks to the IEDP!

Lines 18–19: Pure deletion.

Line 20: The *RELIABILITY* of then is greater than the *INSERT-COST* for the missing parts of the expression ⇒ insertion.

Line 21: *RELIABILITY*(else) > INSERT-COST(then ⟨*statement*⟩) ⇒ insertion.

Lines 20–21: Pure insertion.

Line 22: Combined deletion/insertion. Note: no recovery at WALK in ⟨*statement*⟩—though this would be correct—since *INSERT-COST* of then is too high.

Line 25: *INSERT-COST*(then) > *RELIABILITY*(';') ⇒ no recovery at';'.

Line 27: No recovery at until in ⟨*repeat-statement*⟩ because of costing: *RELIABILITY*(until) < INSERT-COST (to ⟨*expr*⟩ do ⟨*statement*⟩ end).

Line 32: The phrase-parser analyses the misplaced label declaration and detects the ','-error; spurious errors are avoided.

```
    1   program ERROR_DEMO(OUTPUT);
    2
    3       type WEEKDAY = (MO,TU,WE,TH,FR,SA);
    4       var DAY: WEEKDAY;
    5           CAR, DAMAGED, BROKEN, AGE: INTEGER;
    6           NO_GAS, CRASH, VACATION: BOOLEAN;
    7
    8       procedure WORK;       begin      end;
    9       procedure GO_BY_CAR;  begin      end;
   10       procedure WALK;       begin      end;

   12   begin
   13       if CAR = DAMAGED                 then WALK else GO_BY_CAR;
⇒  14       if CAR = DAMAGED                 then WALK ELZE GO_BY_CAR;
⇒  15       IFCAR = DAMAGED                  then WALK else GO_BY_CAR;
⇒  16       if CAR = DAMAGED                 then WALK else GO_BY_CAR;
⇒  17       if BUS  = DAMAGED                then WALK else GO_BY_CAR;
⇒  18       if CAR = DAMAGED ˽ BROKEN        then WALK else GO_BY_CAR;
⇒  19       if CAR = DAMAGED ˽ or NO_GAS     then WALK else GO_BY_CAR;
⇒  20       if CAR =                         then WALK else GO_BY_CAR;
⇒  21       if CAR = DAMAGED                           else GO_BY_CAR;
⇒  22       if CAR = DAMAGED                 SO  WALK else GO_BY_CAR;
   23
   24       repeat
⇒  25           if not VACATION ;
   26             then begin
                        for DAY: = MO until F R do WORK;
   28                   write ('WEEKEND');
   29                 end;
   30       until AGE = 65;
   31
⇒  32       label 1 , ˽ 2;
   33
⇒  34       write ( '_ 0815 )      (* end of comment missing ...
   35   end.
```

(Line 27 is marked ⇒ in the listing.)

```
Line 14: typing-error — 'ELZE' changed to 'else'
Line 15: concatenation-error — 'IFCAR' changed to 'if CAR'
Line 16: transposition-error — 'DAMAGDE' changed to 'DAMAGED'
Line 17: undeclared identifier 'BUS'
Line 18: ', BROKEN' deleted before 'then'
Line 19: ',' deleted before 'or'
Line 20: '<SIMPLE-EXPRESSION>' inserted before 'then'
Line 21: 'then <STATEMENT>' inserted before 'else'
Line 22: 'So WALK' changed to 'then <STATEMENT>'
Line 25: ';' deleted before end of line
Line 27: 'until FR' changed to 'to <EXPRESSION>'
Line 32: misplaced <LABEL-DECLARATION> found
         ',' deleted before '2'
Line 34: second quote missing
               "' deleted before '0815'
               missing end-of-comment. '*)' inserted at end of line
```

Figure 2. Pascal program and error messages

Line 34: Restriction of comments and string constants to one line allows early recovery (language design).

PRACTICAL RESULTS

The quality of a syntactic error recovery method cannot be proved by theoretical considerations, but has to be tested by practical experiments. The Pascal version of our method was tested in 1981 by more than a hundred students for several thousand programs. It was shown that the method indeed produced spurious errors very rarely and overlooked hardly any error. The generated error messages were understandable even to Pascal beginners.

Furthermore, the Pascal parser was tested with the error benchmark developed by Ripley and Druseikis.[11] The material was taken from 237 erroneous Pascal programs written by students (about 12,000 lines). The errors were isolated and provided with minimal context. Identical errors were included only once but provided with a weighting factor. This resulted in 127 program examples which can be obtained from Ripley.

The generated error diagnoses were divided into three categories:

(i) 'Excellent': human diagnosis would produce the same results
(ii) 'good': incorrect parser assumptions, but no spurious errors
(iii) 'poor': one or more spurious or undetected errors

In Table I the method presented here is denoted by the abbreviation BMS. The IBM Pascal/VS-compiler performs an error recovery according to the method developed by Wirth.[3] It produced the result denoted by IBM. The 'global context recovery' developed by Pai and Kieburtz[6] was also evaluated by means of the error benchmark. This yielded the result denoted by GCR.

Table I. Benchmark results

	GCR (%)	IBM (%)	BMS (%)
Excellent	52	68	77
Good	26	9	14
Poor	22	23	9

Consequently the proposed method leads, in 90 per cent of the tested cases, to correct recovery actions, thus avoiding spurious errors. Wrong parser reactions were mainly to be explained through incorrectly used keywords or missing comment brackets and quotes. Such errors present great problems to all methods. Therefore, they should be avoided by an appropriate language design (cf. comments in Ada).

REFERENCES

1. B. W. Pollack (ed).), *Compiler Techniques*, Auerbach Publishers, Philadelphia 1972.
2. J. Feldman and D. Gries, 'Translator writing systems' *CACM*, **11** (2), 77–113 (1968).
3. N. Wirth, 'Die Behandlung von syntaktischen Fehlern', in *Compilerbau*, Teubner-Verlag, 1977.
4. Karl-Rudolf Moll, 'Suffixanalyse, ein Konzept zur Behandlung von syntaktischen Fehlern', *Informatik Spektrum* **4**, 82–89 (1981).

5. C. N. Fischer, D. R. Milton and S. B. Quiring, 'Efficient LL(1) error-correction and recovery using only insertions', *Acta Informatica*, **13** (2), 141–154 (1980).
6. A. B. Pai and R. B. Kieburtz, 'Global context recovery: a new strategy for syntactic error recovery by table-driven parsers', *ACM TOPLAS*, **2** (1), 18–41 (1980).
7. S. L. Graham and S. P. Rhodes, 'Practical syntactic error recovery', *CACM*, **18** (11), 639–650 (1975).
8. S. L. Graham, C. B. Haley and W. N. Joy, 'Practical LR error recovery', *SIGPLAN Notices*, **14** (18), 168–175 (1979).
9. J. Lewi, K. De Vlaminck, J. Huens and M. Huybrechts, 'The ELL(1) parser generator and the error recovery mechanism', *Acta Informatica*, **10**, 209–228 (1978).
10. Johannes Röhrich, 'Behandlung syntaktischer Fehler', *Informatik Spektrum*, **5**, 174–184 (1982).
11. G. D. Ripley and F. C. Druseikis, 'A statistical analysis of syntax errors', *Computer Languages*, **3** (4), 227–239 (1978).
12. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Chap. 11 'Error detection and recovery', Addison-Wesley, 1977, pp. 382–405.
13. C. N. Fischer, K. C. Tai and D. R. Milton, 'Immediate error detection in strong LL(1) parsers', *Information Processing Letters, Vol.* **8** (5), 261–266, (1979).
14. C. Ghezzi, 'LL(1)-grammars supporting an efficient error handling', *Information Processing Letters*, **3** (6), 174–176 (1975).
15. D. Gries, *Compiler Construction for Digital Computers*, Chap. 15 'Error recovery', Wiley, 1971, pp. 314–326.
16. C. Beilken, F. Mattern and M. Spenke, 'Bibliography of error handling in compilers'. University of Kaiserslautern, internal report. The very extensive bibliography (more than 200 titles) contains abstracts and cross references and can be obtained from F. Mattern.