Science of Computer Programming 9 (1987) 271-286
North-Holland

A NEW ERROR REPAIR AND RECOVERY SCHEME FOR LEXICAL AND SYNTACTIC ANALYSIS

Pierre BOULLIER and Martin JOURDAN
INRIA, 78153 Le Chesnay Cedex, France

Communicated by G. Berry
Received October 1985
Revised March 1987

Abstract: This paper describes a two-level error repair and recovery scheme applicable to table-driven parsers and scanners. For parsers, the first level of the scheme tries to locally correct erroneous text by performing insertions, deletions and replacements of tokens around the error detection point, and matching these source modifications against an ordered list of correction models. If this local repair of text fails, a global recovery is initiated, which skips the text up to a "key terminal" and pops the parse stack. Lexical error processing is based on similar principles. The main advantages of the scheme are its power, efficiency and language independence. It can be parameterized by the grammar writer, uses the normal analysis tables and does not slow down the analysis of correct portions of text. Furthermore it can be easily implemented in automatically generated analysers such as the ones constructed by our system SYNTAX.

1. Introduction

The most frustrating and most costly part of the development of a computer application is the debugging process. It is of extreme importance that every compiler provides as much information as possible about compile-time errors, in order to reduce the amount of time spent in debugging. An ideal compiler should

(a) detect *all* errors in a single scan of the source text, and this detection should arise as early as possible;

(b) give a precise, clear and concise message for each error, directing the programmer towards the correction to be done;

(c) avoid introducing non-existent errors (as a consequence of processing an error) and flagging them;

(d) not be slowed down by the existence of a module dealing with errors when analyzing correct programs or correct portions of erroneous programs.

This behaviour must be respected by each module of the compiler having to deal with possible errors: scanner, parser, static semantic checker, etc. In this paper, however, we shall concentrate only on syntactic and lexical errors.

Much of previous and current research effort deals with providing systems which allow to produce automatically lexical and syntactic analysers from high-level language descriptions such as regular expressions and BNF grammars. We have developed such a system, called SYNTAX [1] [3], with enhanced error processing capabilities achieving the goals listed above, plus a further goal, which we feel is as important as the others, and will be the focus of this paper:

(e) the error processing module should be built automatically from those high-level language descriptions, requiring a minimum of extra information from the grammar writer.

SYNTAX is basically an integrated scanner and LR-parser generator which has reached an industrial stage: it has been used in the European Ada Compiler Project [2] and other industrial sites in France and Europe, and in many research projects such as Mentor [6]. It can be associated with many 'semantic processing' techniques, e.g. attribute grammars [13, 17], abstract tree construction [4] and pretty-printers [5]. SYNTAX is implemented in C and runs on UNIX [3].

The error recovery scheme which we present meets the goals listed above and has the following other properties:

- it is powerful, as experience shows, and efficient;
- it is language-independent, but can be parameterized by the grammar writer to achieve a precise "tuning" to the intended use of the generated analyser;
- it uses the normal analysis tables plus only a little of additional information;
- the interface with semantic processing is easy;
- it is mainly independent of the analysis method chosen (as a proof, let us just notice that, although initially devised for syntactic analysis, its application to lexical analysis is straightforward).

Error revovery has long been an attractive research topic (for a good survey, see [11]). Ad-hoc methods have now been formalized and refined to achieve automated construction. However the most powerful methods (e.g. locally least-cost error correction [2]) are still unfortunately the less practically applicable because of their high complexity. One has to find a good tradeoff between power and efficiency.

This paper is organized as follows. We first give a general overview of our error processing scheme (Section 2). Then, in Sections 3 and 4, we detail its two main phases; this presentation will focus on LR syntactic analysis, but adaptations to lexical analysis and other parsing methods are also suggested. The last section is devoted to an example of its capabilities and to comparisons with previous works.

2. General overview of the error processing strategy

The processing of an error in the source text involves three steps:

(a) *Detection*: the error is detected as is usual with table-driven automata, i.e. when the action corresponding to the current state and current input symbol is

[1] SYNTAX is a registered trademark of INRIA.

[2] Supported by the European Economic Community. Ada is a registered trademark of the Government of the U.S.A., AJPO.

[3] UNIX is a registered trademark of AT&T Bell Laboratories.

"error". Note that, since LR methods have the correct prefix property [16], each error is detected exactly when it occurs.

(b) *Display*: SYNTAX provides a facility to display messages exactly where desired, with precise and visible markers. The messages themselves are supplied by the grammar writer (see Section 3).

(c) *Repair and recovery*: after an error occurs, the analyser should try to repair it; whatever the result of this trial, it should be able to restart in a "good" state. This is the topic of this paper.

Repair and recovery occurs in two steps:

- first a *local correction* is attempted: the source text is changed around the error point in order to get a correct text; the analyser is then restarted with this new text as input;
- if this fails, a *global recovery* resembling an enhanced "panic mode" is performed.

These two steps are detailed in the next two sections.

3. Local correction

For this first step of the recovery, the analyser is used as a generator [4]: text portions which are correct suffixes of the text before the error detection point are generated and compared to the original source text and to an ordered list of *correction models* supplied by the grammar writer. These models can specify the deletion, insertion and/or replacement of any number of symbols in the vicinity of the error detection point. The symbols which are taken into account are the symbol in error, the one preceding it and any number of following ones. The source text is represented as

...0 1 2 3 4...

where "1" is the symbol where the error is detected, "0" its predecessor and "2", "3",... its successors. As an example the following model:

0 X 2 3 4

specifies that the symbol in error ("1") is to be replaced by any correct symbol ("X"),

0 2 3 4

specifies that it is to be deleted,

0 X 1 2 3

[4] This generation starts from the current parser state, and thus uses the accumulated left context represented by the parse stack.

that a symbol is to be inserted before it, and

1 2 3 4

that the preceding symbol ("0") is to be deleted.

Virtually, given a correction model, the automaton produces all correct sequences of symbols of the length of the model and tries to match them with the model: symbols corresponding to numbers in the model should match the corresponding ones in the source text, and "X" symbols match any token (see the algorithm in Fig. 1). If this matching succeeds, the erroneous portion of text is replaced by the sequence thus produced, and the analyser can start again. Since this new sequence is correct, no subsequent error will be detected up to its end; thus "unchanged" symbols at the tail of the model (2 3 4 in the first one, for instance) act as a *validation part* for the correction: the corresponding tokens in the original text must be correct to validate the trial.

It should be noticed that the abstract algorithm presented in Fig. 1 can be efficiently implemented: each call to "Parse(sm[1 ... k])" in Match (lines 20 and 23) has only to test whether "sm[k]" is correct since we know that "sm" has already been validated on 1 ... k − 1. The loop "for each terminal t" (line 18) can be implemented as "for each terminal t which is syntactically correct w.r.t. the current parse stack", where "current" means after the parse of "sm(1 ... k − 1]"; the terminals are those for which the parsing action in the current state is not "error". One also has to take care of making "Parse" not destroy the original parse stack P_0; this can be achieved without copy by noting at each transition (e.g. into a local variable if Parse is implemented as a recursive procedure) the modifications affecting the parse stack and undoing them upon return. The global worst-case complexity of this algorithm is thus exponential in the number of "X"s in the compound string of the correction models (and linear in the rest of the string), but experience shows that this combinatorial explosion does not occur in practice; furthermore, it seems that users allow a system to consume some time if the result is worth the effort, that is if the correction is good (and user friendly!).

This abstract algorithm can in principle be applied to any system in which the analyser has an operation mode which allows it to answer *yes* or *no*, given an input string and a start state. Table-driven analysers obviously fulfill this condition, but it also seems possible—although we did not try it—to modify recursive-descent parsers or natural language analysers to achieve it. Thus this algorithm has a very broad application field.

The length of each model, the number of insertions, deletions and/or replacements performed and the validation length are unbounded; but it is obvious that the efficiency of the recovery will strongly depend on them, and a good tradeoff has to be found for each language.

The list of correction models is given in order of decreasing priority: the correction is attempted according to each model in turn, starting from the first and ending as soon as a valid correction is found.

```
{ The input token stream is described as ...0 1 2 3 4 5... where
  "1" is the error detection point.
  Let P0 be the parse stack before the transition upon the "0"
  token. The boolean function "Parse(s)" returns true if "s" is
  a correct suffix w.r.t. P0.
  To each correction model "m" is associated a token string "sm" of
  length |m| which will contain the corrected text if model "m"
  succeeds. }

procedure LocalCorrection (Success: out boolean;
                           CorrectedText: out TokenString) ;
begin
   for each model m in decreasing priority order do
      allocate sm of length |m| ;
      for each i in 1..|m| do
         if m[i] = "X" then
            sm[i] := EmptyToken
         else { m[i] = a number }
            sm[i] := Input[m[i]]
         fi
       od ;
      if Match (1, m, sm) then
         { As a side effect, Match will replace the EmptyTokens
           in sm by actual tokens. }
         Success := true ;
         CorrectedText := sm ;
         return
      fi
   od ;
   Success := false;
   return
end LocalCorrection;

function Match (k:  in integer ;
               m:  in CorrectionModel ;
               sm: in out TokenString) : boolean;
{ "m" is the model being tried ; the string "sm" has already been
  validated on length k-1. }
begin
   if (for each i ≥ k, m[i] ≠ "X") then
      { the above condition is static once you know the models }
      { check the validation part }
      return Parse(sm)
   elsif m[k] = "X" then
      for each terminal t do
         sm[k] := t;
         { insert a terminal token... }
         if Parse(sm[1..k]) and then Match(k+1, m, sm) then
            { ...and check it }
            return true
         fi
      od ;
      return false
   else
      { check the specified input symbol and continue }
      return Parse(sm[1..k]) and then Match(k+1, m, sm)
   fi
end Match ;
```

Fig. 1. Local correction algorithm.

This error repair scheme is used in both the (table-driven) scanner and parser of SYNTAX; more details on lexical error correction appear at the end of this section. For syntactic analysis this local correction is enhanced with a facility to correct spelling errors on keywords [19]. The possible corrections are insertion, deletion or replacement of a single character, and exchange of two contiguous characters. A model describing such a correction is

```
0 S 2
```

which specifies that the token in error might be a misspelled keyword.

The list of default correction models supplied by SYNTAX is given in Fig. 2, under the headings "Local". Notice that it is only a part of a complete specification of all aspects of error handling (as defined in Section 2).

The error messages corresponding to each correction model are also supplied by the grammar writer (see Fig. 2) as a sequence of texts (constant strings a la C), "dollar" symbols and "percent" symbols. The symbols "$0", "$1", etc., will be replaced by the text of the original tokens in the source and "%0", "%1", etc., will be replaced by the text of the "new" tokens (if relevant). These messages are thus given only in terms of the source text, and do not refer to the grammar itself which should remain unknown to the end user[5]. Notice also that no (portion of) message is built-in in the system; this allows complete adaptation to any natural language (English, French,...).

A correction is flagged as a "warning" (the actual "title" can be redefined, see Fig. 3) if it involves no insertion of *generic terminals*, i.e. terminals having a variable text such as identifiers or numbers; else it is flagged as an "error". Indeed, if this local correction succeeds, the parse tree will be correct in any case, but inserting a generic terminal has no semantic meaning. With SYNTAX an inserted generic terminal will have an empty text, and the semantic routines must be designed to handle this situation.

The possibility to act on the token preceding the error detection point is very useful in conjunction with the spelling error correction. Consider the following erroneous Pascal phrase:

```
if ... then begun x := ...
```

The error detection point is the token "x", because "begun" is parsed as an identifier which is a legal follower of "then"; however the clear intent of the programmer was to write "begin" instead of "begun". Using model "S 1" and spelling correction, "begun" (token "0") is replaced by "begin".

[5] The only exception to this rule occurs when a generic terminal (see the definition in the next paragraph of the main text) is to be inserted or is to replace an original token; in this case the corresponding "percent" symbol appears in the message as the name of that generic terminal as defined by the grammar writer (e.g. "%IDENTIFIER", "%NUMBER", etc.). They should therefore be meaningful. See Fig. 3 (line 5) for an example.

```
Titles
   "",
   "Warning:\t",
   "Error:\t";

Scanner
    Local
      1 2 3 4   ; "The invalid character \"" $0 "\" is deleted.";
      X 1 2 3 4 ; "The invalid character \"" $0 "\" is replaced by \"" %0 "\".";
      X 0 1 2 3 ; "The character \"" %0 "\" is inserted before \"" $0 "\".";

      Dont_Delete = {};
      Dont_Insert = {};

   Global
       Detection  : "\"%s\" is deleted.";
      -- parameter: character in error
       Keyword    : "This unknown keyword is erased.";
      Eol        : "End Of Line";
      Eof        : "End Of File";
      Halt       : "Scanning stops on End Of File.";

Parser
    Local
      0 S 2     ; "Misspelling of \"" $1
                "\" which is replaced by the keyword \"" %1 "\".";
      S 1       ; "Misspelling of \"" $0 "\" before \"" $1
                "\" which is replaced by the keyword "\" %0 "\".";
      0 X 1 2 3 ; "\"" %1 "\" is inserted before "\" $1 "\".";
      0 X 2 3 4 ; "\"" $1 "\" is replaced by \"" %1 "\".";
      0 2 3 4   ; "\"" $1 "\" is deleted.";
      0 X X 1 2 3 4 ;
                "\"" %1 " " %2 "\" is inserted before \"" $1 "\".";
      X 0 1 2 3 ; "\"" %0 "\" is inserted before \"" $0 " " $1 "\".";
      X 1 2 3 4 ; "\"" $0 "\" before \"" $1 "\" is replaced by \"" %0 "\".";
      1 2 3 4   ; "\"" $0 "\" before \"" $1 "\" is deleted.";
      X 2 3 4   ; "\"" $0 " " $1 "\" is replaced by \"" %0 "\".";
      X X 1 2 3 ; "\"" $0 "\" before \"" $1 "\" is replaced by \""
                %0 " " %1 "\".";

      Dont_Delete = {}; -- depends on the language...
      Dont_Insert = {};

   Forced Insertion
      "\"" %0 "\" is inserted before \"" $1 "\"." ;

   Global
      Key_Terminals = {}; -- includes EOF
      Validation Length = 2;
      Followers Number <= 5 : "\"%s\"^(, \"%s\"^) is expected.";
      -- parameters: array [1:FollowersNumber] of valid followers
      -- at detection point
      Detection  : "Global recovery.";
      -- parameters: none
      Restarting : "Parsing resumes on "\"%s\"";
      -- parameters: array [1:ValidationLength] of valid followers
      -- at restarting point
      Halt       : "Parsing stops on End Of File.";
      -- parameters: none

Abstract
   "%d errors and %d warnings are reported.";
   -- parameters: array [1:TitlesNo] of number of messages
```

Fig. 2. Standard error handling specification.

This possibility implies that we may have to modify already parsed portions of text (the "0" token); in order to avoid "undoing" the semantic actions that could have been executed upon reductions performed on this token, we maintain two stacks, the "analysis" stack and the "semantic actions" stack, the latter being "one token late" w.r.t. the former. The management of these two stacks is the only point which slows the parsing of correct portions of text, by about 15%.

The local correction at the syntactic level is improved by two other features:

- the grammar writer may define a set of *key terminals*; those terminals should be high-level terminators or separators such as ";" or "END" in Pascal; their main use is in the second error recovery level (see next section), but they also help the local correction as follows: when processing a model, the correction will be validated either at the end of the model or at the first symbol which corresponds to a key terminal in the source text; this allows for instance local corrections at the end of a statement even if there is another error at the beginning of the next statement;
- the grammar writer may also define the sets of *don't delete* (DD) and *don't insert* (DI) terminals; their meaning is the following: if the first valid correction (with higher priority) involves the deletion of a DD-terminal or the insertion of a DI-one, then we check the models with a lower priority; if one of them corresponds to a valid correction which does not violate the DD-DI rule, then we apply this one; if no other local correction is possible, then the first one, which does violate this rule, is accepted.

The DD- and DI-terminals sets should both contain "opening" symbols which are to be matched by a corresponding "closing" symbols; among such pairs let us quote the Pascal symbols "begin" . . . "end", "(". . .")" and "repeat" . . . "until". One should not insert an opening symbol because it is highly probable that the corresponding closing one would have to be inserted (spurious error); similarly, if one detetes an opening symbol, it is highly probable that the closing one would have to be deleted. The DD-terminals set should also contain unambiguous symbols which have a very high level meaning, such as "procedure", "function" or even "array" in Pascal, because their deletion would certainly cause a large number of spurious errors. For further discussion about DD- and DI-terminals, see section 5.

If no correction model matches, and the analyser is in a state such that only one symbol is valid as follower, then this symbol is inserted; this is called *forced insertion.* The parser is then restarted and may discover an error just after the newly inserted terminal, which it will try to correct. An impressive example is a Pascal text in which the whole program header has been forgotten:

```
var x: integer; . . .
```

In this case, "program" is forced before "var", and local correction will insert "%ID" and ";" before "var", using model "0 X X 1 2 3 4".

The error recovery scheme for the lexical level is based on the same principles,

including the DD- and DI-character sets [6], except that
- we do not allow to act on the character preceding the one in error;
- there is no spelling error correction;
- there is no key symbol.

When considering the small number of errors detected at the lexical level, we feel this is sufficient. The standard correction models for lexical analysis are also presented in Fig. 2, together with the corresponding message patterns.

Experience shows that about 80% of the errors in "usual" erroneous programs are corrected by this scheme (see figures in Section 5).

4. Global recovery

If local correction fails, then the error cannot be corrected; but the analyser has still to be restarted in a "good" state. We first discuss *global recovery* for syntactic analysis.

Our global recovery resembles some kind of "panic mode" tailored to LR techniques. Starting from the error detection point, the source text is read without analysis until one of the key terminals (defined in the previous section) is encountered. Then the stack is examined from top to bottom until we find a state having a (valid) transition on a non-terminal which can be followed by this key terminal. Finally the stack is popped to this state, the transition is performed and the analyser is restarted with this new stack and that key terminal as current input symbol. If no such state can be found in the stack, the source text is read up to the next key terminal and the same processing is applied. This scheme always terminates since the end-of-file marker is always considered as a key terminal; the so-called state is then (at worst) the initial state and the non-terminal the start symbol of the grammar.

There is also a validation length associated to this recovery scheme, which can be defined by the grammar writer (see Fig. 2: the standard length is two, i.e. the key terminal itself and the next symbol).

Messages for global recovery are also defined by the grammar writer (see Fig. 2).

When global recovery is performed, the parse tree is incorrect. The semantic routines are supplied with the part of the stack to be popped and thus can react appropriately. For instance, if the actions just construct an abstract syntax tree to be used by subsequent phases, a possible recovery is to collect all the partial trees corresponding to these popped entries as the sons of an "error" node which will take the place of the above defined non-terminal [4].

As described, this global recovery scheme is applicable only to LR methods, but it seems that similar methods exist for LL (e.g. [20]) or recursive descent (e.g. [22]) analysis, although we have not yet formally proved these equivalences.

[6] An example of a DI-character is the beginner of a character string or of a comment.

The two-stacks analysis feature is also used in the global error recovery scheme, in order to avoid undoing the semantic actions associated to reductions performed without checking the lookahead set (the latter being checked only on the next shift action). Those reductions are a consequence of an optimisation which is implemented in almost every LR parser [1], and of course in SYNTAX.

It is important to notice that, in contrast with the local correction scheme, the behaviour of the global recovery is strongly dependent on the grammar itself, and not only on the language and the set of key terminals. Consequently it may sometimes prove useful to tune the grammar to achieve a good global error recovery.

For lexical analysis, the global recovery is to delete the offending character.

5. Evaluation and comparisons

As an example Fig. 3 presents the performance of SYNTAX on the classical erroneous program by Graham and Rhodes [10], a "standard" for papers dealing with error recovery. This program is written in a subset of Algol.

The sets of correction models, key terminals and DD-DI terminals are presented in Fig. 4.

The main point to keep in mind when evaluating this example is that our recovery is purely syntactical, unlike the one in [9], and that we do not do any *forward move* [10]. This explains why no "if" is inserted after the label in line 4: this would have required a forward move [7] to parse the whole expression and discover the "then"; instead, an assignment symbol is inserted after "i" to form the beginning of an assignment statement; a global error recovery occurs thereafter on "then", reducing the entire phrase to ⟨IF_EXPRESSION⟩. The other corrections are "excellent" in the sense defined below and require no comment.

We also evaluated our method on the set of erroneous Pascal programs gathered by Druseikis and Ripley [8] and also used by Penello and DeRemer [21], Graham et al. [9] and Pai and Kieburtz [20]. We used a relatively standard LALR(1) grammar of Pascal, except that it allows multiple declarative parts; as a consequence, only 121 of the original 126 programs were actually erroneous w.r.t. our grammar. Of course we tuned the recovery specification for the given set of programs (but not for each program ...); it is presented in Fig. 5. The tuning consisted mainly in choosing the right order for the correction models, and the right sets of key-, DD- and DI-terminals. The addition of nearly all the operators into the DI set stems from the fact that our algorithm cannot choose which operator to insert at a given point, if it has to insert one; we decided to have only "+" and "=" as possible insertions, since those operators have the largest application field and produce the fewer semantic error messages.

[7] This could have been done on this particular example by local correction if the model "X 0 1 ..." had had a higher priority than "0 X 1 ...", but this rather far-fetched, and would have failed anyway if "i" had been replaced by a more complicated ⟨VARIABLE⟩ such as "i(x)".

```
     1 :   begin
     2 :    integer array A,B(1..5 1..10);
                                  ↑

**** Warning :     "," is inserted before "1".

     3 :    integer I,J,K,L;
     4 :     UP :I+J > K+L*4 then go L1 else K is 2;
                ↑          ↑       ↑         ↑
                0          1       2         3

**** Warning (0) : ":=" is inserted before "+".
**** Error (1) :   Global recovery.
**** Warning (1) : Parsing resumes on "then".
**** Warning (2) : "to" is inserted before "L1".
**** Warning (3) : "is" is replaced by ":=".

     5 :     A 1,2 := B(3*(I + 4, J*/K)
               ↑   ↑            ↑   ↑
               0   1            2   3

**** Warning (0) : "(" is inserted before "1".
**** Warning (1) : ")" is inserted before ":=".
**** Warning (2) : ")" is inserted before ",".
**** Error (3) :   "%IDENTIFIER" is inserted before "/".

     6 :     if I = 1 then then go to UP ;
             ↑             ↑
             0             1

**** Warning (0) : ";" is inserted before "if".
**** Warning (1) : "then" is deleted.

     7 :     L2: end

          2 errors and 10 warnings are reported.
```

Fig. 3. An example of error processing by SYNTAX.

The results of the evaluation are presented in Fig. 6. We classified the accuracy of the recovery globally for each program, not for each error. We considered the recovery as "excellent" when the correction performed was the one a human reader might have made [8]; it was counted as "mean" when the corrected program had some meaning, but not the one which the writer "intended"; and it was counted as "poor" in any other case. Note that this is the classification also used by Pai and Kieburtz [20]. Since it was sometimes difficult to classify a given behaviour into a category rather than into an other one, two sets of figures are given: an "optimistic" one, in which, in case of doubt, the best classification was chosen, and a pessimistic one. When remembering that our recovery scheme is purely syntactical, one can see that these figures are not so bad.

[8] The "optimal" correction is generally given in the program itself as a comment [8].

```
Titles
   "",
   "***** Warning:\t",
   "***** Error:\t";

Scanner
    Local
      1 2 3 4   ; "\"" $0 "\" is deleted.";
      X 1 2 3 4 ; "\"" $0 "\" is replaced by \"" %0 "\".";
      X 0 1 2 3 ; "\"" %0 "\" is inserted before \"" $0 "\".";

      Dont_Delete = {};
      Dont_Insert = {};

   Global
       Detection  : "\"%s\" is deleted.";
       Keyword    : "This unknown keyword is erased.";
       Eol        : "End Of Line";
       Eof        : "End Of File";
       Halt       : "Scanning stops on End Of File.";

Parser
    Local
      0 S 2     ; "Misspelling of \"" $1
                "\" which is replaced by the keyword \"" %1 "\".";
      S 1       ; "Misspelling of \"" $0
                "\" which is replaced by the keyword \"" %0 "\".";
      0 2 3 4   ; "\"" $1 "\" is deleted.";
      0 X 1 2 3 ; "\"" %1 "\" is inserted before \"" $1 "\".";
      0 X 2 3 4 ; "\"" $1 "\" is replaced by \"" %1 "\".";
      0 X X 1 2 3 4 ;
                "\"" %1 " " %2 "\" is inserted before \"" $1 "\".";
      X 0 1 2 3 ; "\"" %0 "\" is inserted before \"" $0 " " $1 "\".";
      X 1 2 3 4 ; "\"" $0 "\" is replaced by \"" %0 "\".";
      1 2 3 4   ; "\"" $0 "\" is deleted.";
      X 2 3 4   ; "\"" $0 " " $1 "\" is replaced by \"" %0 "\".";
      X X 1 2 3 ; "\"" $0 "\" is replaced by \"" %0 " " %1 "\".";

      Dont_Delete = {"begin", "end"};
      Dont_Insert = {"(", "begin"};

   Forced  Insertion
      "\"" %0 "\" is inserted before \"" $1 "\"." ;

   Global
      Key Terminals = {";", "then", "else", "end"};
      Validation Length = 2;
      Followers Number <= 5 : "\"%s\"^(, \"%s\"^) is expected.";
      Detection  : "Global recovery.";
      Restarting : "Parsing resumes on "\"%s\".";
      Halt       : "Parsing stops on End Of File.";

Abstract
   "%d errors and %d warnings are reported.";
```

Fig. 4. Error handling specification for a subset of Algol.

```
Titles
   "",
   "Warning:\t",
   "Error:\t";

Scanner
    Local
      1 2 3   ; "The invalid character \"" $0 "\" is deleted.";
      X 1 2 3 ; "The invalid character \"" $0 "\" is replaced by \""
                %0 "\".";
      X 0 1 2 ; "The character \"" %0 "\" is inserted before \"" $0 "\".";

      Dont_Delete = {};
      Dont_Insert = {"{", "\""};

   Global
       Detection  : "\"%s\" is deleted.";
       Keyword    : "This unknown keyword is erased.";
       Eol        : "End Of Line";
       Eof        : "End Of File";
       Halt       : "Scanning stops on End Of File.";

Parser
    Local
      0 S 2      ; "Misspelling of \"" $1
                 "\" which is replaced by the keyword \"" %1 "\".";
      S 1 2      ; "Misspelling of \"" $0 "\" before \"" $1
                 "\" which is replaced by the keyword \"" %0 "\".";
      0 X 1 2 3 ; "\"" %1 "\" is inserted before \"" $1 "\".";
      0 X 2 3 4 ; "\"" $1 "\" is replaced by \"" %1 "\".";
      0 2 3 4    ; "\"" $1 "\" is deleted.";
      0 X X 1 2 3 4 ;
                 "\"" %1 " " %2 "\" is inserted before \"" $1 "\".";
      X 0 1 2 3 ; "\"" %0 "\" is inserted before \"" $0 " " $1 "\".";
      X 1 2 3    ; "\"" $0 "\" before \"" $1 "\" is replaced by \"" %0 "\".";
      1 2 3 4    ; "\"" $0 "\" before \"" $1 "\" is deleted.";
      X 2 3 4    ; "\"" $0 " " $1 "\" is replaced by \"" %0 "\".";
      X X 1 2 3 ; "\"" $0 "\" before \"" $1 "\" is replaced by \""
                 %0 " " %1 "\".";

      Dont Delete = {"BEGIN", "RECORD", "END", "IF", "ELSE", "THEN", "FOR",
                     "PROCEDURE", "FUNCTION", "UNTIL", "%ID", "FORWARD",
                     "WRITE", "WRITELN", "["};
      Dont_Insert = {"(", "[", ".", "BEGIN", "RECORD", "CASE", "REPEAT",
                     "IF", "WITH", "@", "*", "/", "<>", "<", "<=", ">", ">=",
                     "DIV", "MOD", "AND", "OR", "NOT", "THEN"};

   Forced  Insertion
      "\"" %0 "\" is inserted before \"" $1 "\"." ;

   Global
      Key Terminals = {";", ")", "DO", "ELSE", "END", "THEN", "UNTIL"};
      Validation Length = 2;
      Followers Number <= 5 : "\"%s\"^(, \"%s\"^) is expected.";
      Detection  : "Global recovery.";
      Restarting : "Parsing resumes on "\"%s\".";
      Halt       : "Parsing stops on End Of File.";

Abstract
   "%d errors and %d warnings are reported.";
```

Fig. 5. Error handling specification for Pascal.

	Excellent	Mean	Poor
Optimistic	91 (75.2%)	24 (19.8%)	6 (5.0%)
Pessimistic	82 (67.8%)	28 (23.1%)	11 (9.1%)

Lexical Errors	Lexical + Local	Local Correction	Forced Insertion	Global Recovery	Total
25	4	164	9	24	226
(11%)	(1.8%)	(72.6%)	(4%)	(10.6%)	(100%)

Fig. 6. Evaluation of the power of our scheme on a set of Pascal programs.

Figure 6 also shows the relative effect of the different ways to catch, repair and recover from an error; note that this classification is done on an error by error basis, rather than on a program by program basis. The column "Lexical+Local" counts the errors which are corrected by both the scanner and the local correction scheme: for instance, in the Pascal phrase

c[i]'=' ';

the scanner deletes the first quote because it cannot legally follow the closing bracket, and the parser replaces the resulting "=" by an assignment statement. The results show that most errors are repaired by local correction. Indeed, all the "excellent" recoveries involved local correction or forced insertion. Since global recovery does not repair the erroneous text, is cannot lead to "excellent" results.

Let us recall the figures obtained by other authors on the same set of programs. In [20], the results are as follows: 52% excellent, 26% good, 22% poor; in [21]: 42% excellent, 28% good, 30% poor or unrepaired. The comparison with our figures show that the main effect of our error handling scheme is a very important decrease of the proportion of poor repairs, which is probably what the users want.

When comparing our method with others which appeared in the literature, one can make the following remarks:

- it is a good tradeoff between power (it is nearly as good as locally least-cost correction [2]) and efficiency in space and time (no backward move nor forward move [10, 18, 21]);
- it uses the normal parse tables, unlike [7, 21, 9] and even optimised ones [1, 3] thanks to the two-stacks feature;
- using error recovery does not imply to rewrite nor augment the original grammar (error productions in [9]);
- the error recovery module is thus generated automatically and requires no complicated interface with the semantic routines (unlike [9], where the syntactic error recovery is tuned to the language through semantic actions);
- the grammar writer has a large and easy control over the error recovery process, through the correction models and the sets of key-, DD- and DI-terminals.

Our error repair and recovery scheme originates in a number of works. The idea to drive local corrections through models, instead of obscure "weights" [9, 10], was first devised by LaFrance [14]. The spelling error correction is borrowed from Morgan [19]. The global error recovery principle is an adaptation and a generalisation of the ideas of Leinius [15] and James [12].

6. Conclusion

We have presented a two-level error repair and recovery scheme directly applicable to table-driven LR parsers and finite state automaton scanners, and easily adaptable to table-driven LL parsers and probably other analysers. This scheme has a number of advantages over other methods presented earlier.

The presence of this error recovery scheme is one of the main advantages of SYNTAX over other systems having the same pretentions. It has been heavily used in France and in Europe for now more than seven years, and seems entirely satisfactory.

Acknowledgment

Thanks go to Susan Graham, who provided us with a copy of the Druseikis-Ripley set of erroneous Pascal programs. Thanks also go to Bernard Lorho, Philippe Deschamp and Harald Ganzinger and the referees for their useful comments on a draft of this paper. Lastly special thanks go to Antoine Rizk for proofreading the final draft.

References

[1] A.V. Aho and J.D. Ullman, *The Theory of Parsing, Translation and Compiling* (Prentice-Hall, Englewoods Cliffs, NJ, 1972).

[2] S.O. Anderson, R.C. Backhouse, E.H. Bugge and C.P. Stirling, An assessment of locally least-cost error recovery, *Comput. J.* **20** (1983) 15-24.

[3] P. Boullier, Ph. Deschamp and B. Lorho, The SYNTAX reference manual, INRIA, Rocquencourt, 1987.

[4] P. Boullier and K. Ripken, Building an Ada compiler following meta-compilation methods, Séminaires INRIA "Langages et Traducteurs" 1978-1981, INRIA, Rocquencourt, 1981, pp. 99-140.

[5] Ph. Deschamp and P. Boullier, Paradis, un Système de paragraphage dirigé par la syntaxe, Report RR-455, INRIA, Rocquencourt, 1985 (in French).

[6] V. Donzeau-Gouge, G. Huet, G. Kahn and B. Lang, Programming environments based on structured editors: The Mentor experience, Report RR-26, INRIA, Rocquencourt, 1980.

[7] F.C. Druseikis and G.D. Ripley, Error recovery for simple LR(k) parsers, *Proc. 1976 ACM Annual Conference*, Houston, TX (1976) 396-400.

[8] F.C. Druseikis and G.D. Ripley, A statistical analysis of syntax errors, *Comput. Languages* **3** (1978) 227-240.

[9] S.L. Graham, C.B. Haley and W.N. Joy, Practical LR error recovery, *SIGPLAN Notices* **14**(8) (1979) 168-175.

[10] S.L. Graham and S.P. Rhodes, Practical syntactic error recovery, *Comm. ACM* **18**(11) (1975) 639-650.

[11] K. Hammond and V.J. Rayward-Smith, A survey on syntactic error recovery and repair, *Comput. Languages* **9**(1) (1984) 51-67.

[12] L.R. James, A syntax directed error recovery method, Ma. thesis, Report CSRG-13, University of Toronto, Toronto, 1972.

[13] M. Jourdan, Strongly non-circular attribute grammars and their recursive evaluation, *SIGPLAN Notices* **19**(6) (1984) 81-93.

[14] J. LaFrance, Syntax directed error recovery for compilers, PhD thesis, Report 459, Department of Computer Science, University of Illinois, Urbana Champaign, IL, 1971.

[15] R.P. Leinius, Error detection and recovery for syntax-directed compiler systems, PhD thesis, Department of Computer Science, University of Wisconsin, 1970.

[16] J.P. Levy, Automatic correction of syntax errors in programming languages, PhD thesis, Report TR71-116, Department of Computer Science, Cornell University, Ithaca, NY, 1971.

[17] B. Lorho, Semantic attributes processing in the system DELTA, in: Ershov and Koster, Eds., *Methods of Algorithmic Language Implementation*, Lecture Notes in Computer Science **47** (Springer, Berlin, 1977) 21-40.

[18] M.D. Mickunas and J.A. Modry, Automatic error recovery for LR parsers, *Comm. ACM* **21**(6) (1978) 459-465.

[19] H.L. Morgan, Spelling correction in system programs, *Comm. ACM* **13**(2) (1970) 90-94.

[20] A.J. Pai and R.B. Kieburtz, Global context recovery: a new strategy for syntactic error recovery by table-driven parsers, *ACM Trans. Programming Languages and Systems* **2**(1) (1980) 18-41.

[21] T.J. Penello and F.L. deRemer, A forward move algorithm for LR error recovery, *Proc. 5th ACM Symposium on Principles of Programming Languages*, Tucson, AZ (1978) 241-254.

[22] N. Wirth, *Algorithms + Data Structures = Programs* (Prentice-Hall, Englewoods Cliffs, NJ, 1976).