Efficient and Comfortable Error Recovery in Recursive Descent Parsers

J. Grosch

GESELLSCHAFT FÜR MATHEMATIK
UND DATENVERARBEITUNG MBH

FORSCHUNGSSTELLE FÜR
PROGRAMMSTRUKTUREN
AN DER UNIVERSITÄT KARLSRUHE

Project

Compiler Generation

Efficient and Comfortable Error Recovery in Recursive Descent Parsers

Josef Grosch

Dec. 11, 1989

Report No. 19

Gesellschaft für Mathematik und Datenverarbeitung mbH
Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1
D-7500 Karlsruhe

Efficient and Comfortable Error Recovery in Recursive Descent Parsers

Josef Grosch
GMD Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1, D-7500 Karlsruhe, Germany
Tel: +721-6622-26
E-Mail: grosch@karlsruhe.gmd.de

Abstract

This paper describes the interesting features of the recursive descent parser generator *Ell* from the user's point of view. Some of the interesting implementation aspects are discussed. The generated parsers are extremely fast and run at speed of 55,000 tokens per second or 900,000 lines per minute on a MC 68020 processor. The outstanding features of *Ell* are the L-attribution mechanism, its ability to handle non LL(1) grammars, and its comfortable error handling, which includes error reporting, recovery, and repair.

1. Introduction

Recursive descent parsing is an established technique for the analysis of LL(1) languages since many years. It can be implemented easily by a hand-written program or by using one of the existing parser generators such as [DuW81, Gra88a, ReM85]. In real life applications, issues such as good quality error recovery and high run time performance are essential. These requirements are rarely discussed in the literature and seldom achieved by the existing parser generators. This explains the reasons for implementing yet another LL(1) parser generator and to describe some details of the generated code.

This paper presents how comfortable features like error recovery, error repair, and L-attribution are implemented in the high performance parsers generated by the parser generator *Ell* [Gro88, GrV88]. *Ell* generates recursive descent parsers from LL(1) grammars given in extended BNF. The grammar rules may be associated with semantic actions consisting of arbitrary statements which are executed whenever they are passed during a left-to-right parse. An L-attribution may be evaluated during parsing. *Ell* offers possibilities to resolve the LL(1) conflicts of non LL(1) grammars. The generated parsers automatically include error reporting, recovery, and repair. Parsers can be generated in the target languages C and Modula-2. The parsers are extremely fast and run at speed of 55,000 tokens per second or 900,000 lines per minute on a MC 68020 processor.

This paper addresses only the interesting features of *Ell*. The mechanism for L-attribution and the error recovery are described from the user's point of view as well as from the implementation view. We also present how LL(1) conflicts are resolved and discuss further implementation issues such as testing for set membership or the pitfalls of CASE (switch) statements. The examples are taken from a parser for Modula-2 and also use Modula-2 as target language.

2. L-Attribution

According to [Wil79] an attribute grammar which can be evaluated during LL(1)-parsing is called an L-attributed grammar. The notion L-attribution means that all attributes can be evaluated in a single top-down left-to-right tree walk.

2.1. User's View

The specification language of *Ell* distinguishes three kinds of grammar symbols: nonterminals, terminals, and literals. Literals are similar to terminals and are denoted by strings.

Terminals and nonterminals are denoted by identifiers. Terminals and nonterminals can be associated with arbitrary many attributes of arbitrary types. The computation of the attribute values takes place in the semantic action parts of a rule. The attributes are accessed by an attribute designator which consists of the name of the grammar symbol, a dot character, and the name of the attribute. As several grammar symbols with the same name can occur within a rule, the grammar symbols are denoted unambiguously by appending numbers to their names. The left-hand side symbol always receives the number zero. For every (outermost) alternative of the right-hand side, the symbols with the same name are counted starting from one.

Example:

```
expr    : ( [ '+' ] term { expr0.value :=   term1.value;               }
          | '-' term     { expr0.value := - term2.value;               }
          )
          ( '+' term     { INC (expr0.value, term3.value);             }
          | '-' term     { DEC (expr0.value, term4.value);             }
          ) *
        .
term    : fact           { term0.value := fact1.value;                 }
          ( '*' fact     { term0.value := term0.value * fact2.value;   }
          | '/' fact     { term0.value := term0.value DIV fact3.value; }
          ) *
        .
fact    : const          { fact0.value := const1.value;                }
        | '(' expr ')'   { fact0.value := expr1.value;                 }
        .
```

The above example specifies the evaluation of simple arithmetic expressions using one attribute called *value*. The operator "*" denotes repetition zero, once, or more times. The attributes have to be declared as members of a record type called *tParsAttribute*.

Example:

```
TYPE tParsAttribute = RECORD value: INTEGER; END;
```

2.2. Implementation

The implementation of the L-attribution in the generated parsers is very simple. As usual, every nonterminal is analyzed by a procedure. Every procedure has one (reference) parameter referring to the left-hand side attributes. As all attributes are declared as members of one (record) type, one parameter suffices to pass an arbitrary number of attributes. For all right-hand side symbols with attributes, local variables are declared. This solution provides very efficient stacking of attributes via the usual procedure call mechanism. Stacking is necessary for the attribute evaluation of recursive grammar rules.

Example:

```
PROCEDURE expr (...; VAR expr0: tParsAttribute);
VAR term1, term2, term3, term4: tParsAttribute;
BEGIN
   ...
END expr;
```

3. Non LL(1) Grammars

Sometimes grammars do not obey the LL(1) property. They are said to contain LL(1) conflicts. A well-known example is the dangling-else problem of Pascal: in case of nested it-then-else statements it may not be clear to which IF an ELSE belongs. It is very easy to solve this conflicts in hand-written solutions. *Ell* handles LL(1) conflicts in the following ways:

- Several alternatives (operator |) cause a conflict if their FIRST sets are not disjoint: the alternative given first is selected.
- An optional part (operators [] and *) causes a conflict if its FIRST set is not disjoint from its FOLLOW set: the optional part will be analyzed because otherwise it would be useless.
- Parts that may be repeated at least once cause a conflict if their FIRST and FOLLOW sets are not disjoint (as above): the repetition will be continued because otherwise it would be executed only once.

With the above rules it can happen that alternatives are never taken or that it is impossible for a repetition to terminate for any correct input. These cases as well as left recursion are considered to be serious design faults in the grammar and are reported as errors. Otherwise LL(1) conflicts are resolved as described above and reported as warnings.

4. Error Recovery

4.1. User's View

The generated parsers include information and program code to handle syntax errors completely automatically and provide expressive error reporting, recovery, and repair. Every incorrect input is "virtually" transformed into a syntactically correct program with the consequence of executing only a "correct" sequence of semantic actions. Therefore the following compiler phases like semantic analysis don't have to bother with syntax errors. *Ell* provides a prototype error module which prints messages as shown in Figure 1. Appendix 4 contains a larger example demonstrating the behaviour of our method. Internally the error recovery works as follows:

- The location of the syntax error is reported.
- If possible, the tokens that would be a legal continuation of the program are reported.
- The tokens that can serve to continue parsing are computed. A minimal sequence of tokens is skipped until one of these tokens is found.
- The recovery location (restart point) is reported.
- Parsing continues in the so-called repair mode. In this mode the parser behaves as usual except that no tokens are read from the input. Instead a minimal sequence of tokens is synthesized to repair the error. The parser stays in this mode until the input token can be accepted. The synthesized tokens are reported as inserted symbols. The program can be

Source Program:

```
MODULE test;
BEGIN
   IF (a = ] 1 write (a) END;
END test.
```

Error Messages:

```
3, 12: Error       syntax error
3, 12: Information expected symbols: Ident Integer Real String '(' '+' '-' '{' 'NOT'
3, 14: Information restart point
3, 16: Error       syntax error
3, 16: Information restart point
3, 16: Repair      symbol inserted : ')'
3, 16: Repair      symbol inserted : 'THEN'
```

Fig. 1: Example of Automatic Error Messages

regarded as repaired, if the skipped tokens are replaced by the synthesized ones. Upon leaving repair mode, parsing continues as usual.

4.2. Implementation

During LL(1) analysis the following kinds of syntax errors can occur: at the analysis of a terminal the current (look-ahead) token can be different from the expected terminal. At the analysis of alternatives the current token can be member of none of the FIRST sets of the possible branches. At the analysis of optional or iterated parts the current token can be member of neither the FIRST set nor the FOLLOW set of the construct. In order to achieve good quality error recovery, the latter test has to be performed before the analysis of an optional part and before every iteration. This section discusses the interesting aspects of error recovery: how the sets of expected tokens are defined, how parsing is continued after syntax errors, how error repair works, and how an efficient implementation is achieved.

4.2.1. Expected Symbols

At the location of a syntax error, we try to report the set of expected tokens. To be able to report the exact set of expected tokens, syntax errors have to be detected as early as possible. Furthermore, information must be maintained during parsing because the exact sets depend on the dynamic call hierarchy. Early detection of errors requires the knowledge of the exact FOLLOW sets which also depend on the call hierarchy. In order to gain efficiency, we report only the subset of the expected tokens which can be computed at generation time. This set is necessary for every potential error location. In case of a terminal the expected token is just this terminal. In case of alternatives the set of expected tokens is the union of the FIRST sets of the alternatives. In case of optional or iterated parts the set of expected tokens is the union of the FIRST set and of the local FOLLOW set of the construct. All the sets are computed at generation time and stored in the generated parsers.

4.2.2. Recovery Sets

For every possible syntax error a so-called recovery set is determined containing the tokens where parsing can continue. We present the definition of recovery sets for plain BNF, first. Appendix 3 shows the extension to extended BNF using an attribute grammar formalism. In case of a syntax error the situation is as follows (see Fig. 2):

- Analysis of the productions $p_1, \cdots, p_n$ has started.
- Every production p_i is processed by a procedure called X_i.
- Every procedure X_i calls a procedure X_{i+1} to analyze a nonterminal of the right-hand side.
- Procedure X_n detects an error at position Z.
- The current call hierarchy is $X_1, \cdots, X_n$.

p_1:	X_1	$\rightarrow$	Y_{11}	...		X_2	Y_{1j_1}	...	Y_{1n_1}
p_2:	X_2	$\rightarrow$	Y_{21}	...		X_3	Y_{2j_2}	...	Y_{2n_2}
...									
p_{n-1}:	X_{n-1}	$\rightarrow$	$Y_{n-1,1}$	...	$Y_{n-1,i_{n-1}}$	X_n	$Y_{n-1,j_{n-1}}$	...	$Y_{n-1,n_{n-1}}$
p_n:	X_n	$\rightarrow$	Y_{n1}	...	Y_{ni_n}	Z	Y_{nj_n}	...	Y_{nn_n}

Fig. 2: Situation in case of a syntax error ($p_i \in P, X_i \in N, Y_{ij} \in V, Z \in V$)

Let us concentrate first on the situation locally in production p_n. Parsing could continue at the symbols $Y_{nj_n}, \cdots, Y_{nn_n}$. The set of tokens that allow to continue parsing, or in other words the local recovery set R_n is therefore the union of the FIRST sets of $Y_{nj_n}, \cdots, Y_{nn_n}$:

$$R_n = \bigcup_{k=j_n}^{n_n} FIRST\ (Y_{nk})$$

However, in general there may be no token behind the location of the error which is member of the local recovery set R_n. Therefore, the productions $p_{n-1}, \cdots, p_1$ have to be taken into account, too. Parsing can continue at all symbols not analyzed yet:

$$\begin{array}{lcl} Y_{n-1,j_{n-1}} & \ldots & Y_{n-1,n_{n-1}} \\ & \ldots & \\ Y_{2j_2} & \ldots & Y_{2n_2} \\ Y_{1j_1} & \ldots & Y_{1n_1} \end{array}$$

Therefore in general we need all local recovery sets R_i:

$$R_i = \bigcup_{k=j_i}^{n_i} FIRST\ (Y_{ik}) \quad i = 1, \cdots, n$$

The global recovery set R is the union of all involved local recovery sets:

$$R = \bigcup_{i=1}^{n} R_i$$

The global recovery set R is used to stop skipping of tokens. Tokens are skipped until one is reached that is member of R. This location is called a restart point. This method of recovery terminates because only symbols not yet analyzed are considered as potential restart points. Their processing is contained in the current call hierarchy.

In extreme cases the complete rest of the input is skipped. To guarantee termination of skipping, a special token for end of file (`sEof`) has to be member of R. This is assured by augmenting the grammar by the following rule:

$$p_0\colon X_0 \rightarrow X_1\ \texttt{sEof}$$

where X_1 is the original and X_0 is the new start symbol of the grammar.

The definition of R can be modified in some ways. First, if the symbol Z is a terminal it can be included in R in order to improve the behaviour of error recovery in case of superfluous tokens. Second, it is possible to exclude some elements (except `sEof`) from R without causing the recovery to fail. However, its behaviour becomes more coarse because eventually more tokens are skipped. If only `sEof` remains in R, we arrive at panic mode: after skipping the rest of the input no more errors can be found.

The computation of the global recovery set R can consume a considerable amount of run time. For efficiency reasons, we found the following solution quite satisfactory: The local recovery sets can be computed at generation time. For every right-hand side symbol (or position) a local recovery set is computed and stored. The global recovery set has to be computed at run time, as it depends on the current call hierarchy. The union of the local recovery sets is computed only in the case of an error. As long as there is no error, it suffices to maintain a simple data structure that allows to compute the union on demand.

If the local recovery sets are stored in an array of sets it is enough to know the index of a set in this array. The global restart set is represented as a list of such indices. This list allows to effectively compute the global restart set whenever needed.

A list representing the global restart set is passed via a second parameter to every procedure. In case of an error, the local recovery set is added to the list and then the real set union is accomplished. Before calling another procedure to analyze a nonterminal, the list is extended by a suitable element. The list elements are declared as local variable of the procedures. This way, allocation and deallocation of storage is for free. For details see Fig. 3 as well as Appendix 1.

4.2.3. Error Repair

Every incorrect program is repaired (or transformed) into a syntactically correct program. Error repair is accomplished by skipping tokens as described above and by imaginary inserting tokens. This insertion is realized relatively easy by continuing parsing as nothing would have happened. Whenever a terminal is expected which is different from the current input token, it is reported as inserted. Whenever alternatives are analyzed and the current input token is member of none of the possible FIRST sets, an arbitrary branch, which is non-recursive, is selected. The restriction to non-recursive branches is necessary to guarantee termination. There always exists a non-recursive branch as long as the grammar is reduced. Semantic actions are executed during error repair as usual.

Error recovery and error repair is combined in mainly three procedures to handle literals, terminals, and alternatives or EBNF constructs, respectively (see Appendix 2). To avoid superfluous error messages the parser knows two modes. In *normal* mode errors are reported, tokens are skipped, and the mode is changed to *repair* mode. In *repair* mode neither errors are reported nor tokens are skipped. Instead the inserted tokens are reported. The mode is switched

```
TYPE tUnionPtr  = POINTER TO tUnion;
TYPE tUnion     = RECORD              (* type for list elements              *)
                    GlobalRecoverySet   : tUnionPtr;
                    LocalRecoverySet    : SHORTCARD;
                  END;
                                      (* procedure for a nonterminal         *)
PROCEDURE Module (GlobalRecoverySet: tUnionPtr; VAR Module0: tParsAttribute);
VAR
  Union : tUnion;
  Ident1: tScanAttribute;
  Block1: tParsAttribute;
  ...
BEGIN
  Union.GlobalRecoverySet := GlobalRecoverySet;

  IF Token # sMODULE THEN             (* analysis of a literal               *)
    RecoveryLiteral (sMODULE, 138, GlobalRecoverySet);
  ELSE
    Token := GetToken (); IsRepairMode := FALSE;
  END;

  IF Token # sIdent THEN              (* analysis of an attributed terminal *)
    RecoveryTerminal (sIdent, 138, GlobalRecoverySet, Ident1);
  ELSE
    Ident1 := Attribute;              (* receive attribute from scanner     *)
    Token := GetToken (); IsRepairMode := FALSE;
  END;

  Union.LocalRecoverySet := 57;       (* analysis of a nonterminal           *)
  Block (SYSTEM.ADR (Union), Block1);

END Module;
```

Fig. 3: Scheme of the Code for Error Recovery

back to normal when the analysis of a terminal is successful. Then error recovery is finished and parsing continues normally.

The local variable `Union` is a list element for the representation of the global recovery set. It suffices to append this element to the list once in every procedure. Before calling another procedure to analyze a nonterminal the index of a local recovery set is assigned to the list element (here: 57). The extended list is passed as parameter. The procedures for error recovery called ErrorRecovery, RecoveryTerminal, and RecoveryLiteral are given in Appendix 2. They are called only in case of syntax errors. As errors are considered to be a rare event, these procedures do not have to care about efficiency. Their parameters describe the expected token (here: `sMODULE` and `sIdent`), the index of the local recovery set (here: 138), and the global recovery set. For attributed terminals a fourth parameter specifies the variable receiving the attributes from the scanner. The interface to the scanner consists primarily of the following objects: the procedure GetToken returns the next input token, the global variable Attribute contains the attributes of the current token, and the procedure ErrorAttribute (see Appendix 2) is called by the parser to get attribute values for synthesized tokens.

5. Related Work

Much work has been published about error recovery. We limit this discussion on giving reasons for our solution and mention a few similar methods. The design of our error recovery was guided by the following requirements:

- automatic derivation of error recovery from the grammar
- efficient parsing in terms of run time
- provision of error repair

Efficiency implies a backtrack-free strategy and asks for a clever implementation. Automatic derivation and a backtrack-free strategy imply more or less the definition of recovery sets as given above.

We consider error repair to be important because error recovery should not consider syntactical aspects only. Syntax analysis is usually combined with semantic analysis. Although error repair might not transform an error the way the programmer originally intended, it does transform every erroneous program into a syntactically correct one with the consequence that only syntactically correct information is passed to semantic analysis. This allows great simplifications in the latter because it does not have to care about syntax errors.

The definition for the recovery sets given above is not new as it is somehow inherent in the problem. Similar definitions have been presented previously e. g. by [Iro63, ReM85, SMM84]. Hand-written recursive descent parsers usually implement similar designs [Wir86]. The advantages of our solution are the provision of error repair and the efficient implementation of error recovery. The compiler generators Coco [ReM85] and Coco/R [M\90] use the same strategy for error recovery but do not provide error repair. The efficiency of the parsers generated by Coco/R is comparable to the efficiency of our method. Using Coco/R, error recovery has to be tailored by giving simple directives. *Ell* does not require any user engagement and produces error recovery automatically. Appendix 4 presents the behaviour of our method applied to the example program used in [M\90]. The rather long listing of messages is the direct output of the information provided by the parser. The final layout of the messages can be easily adapted to the ideas of the user.

6. Implementation Issues

Parsing is implemented using recursive procedures as outlined above. The operators of extended BNF are mapped to statements of the target language as given in Appendix 1. In obvious cases simple optimizations are exploited. For example the checks for terminals and literals can be omitted in some cases.

More sophisticated implementation decisions concern the CASE/switch statements and the test for set membership. If C is used as target language, it turned out that many C compilers optimize switch statements in favour of storage. If the set of case labels is non-compact, a sorted list of values and addresses is generated. A run time system routine performs binary search in this table in order to map the current switch value to the address of a case branch. To trick the C compiler, *Ell* inserts dummy labels to make the set of case labels compact. Then the C compilers use a jump table which executes considerably faster. Generating compact sets of case labels improved the over all run time of the parsers by 30%.

Sets are implemented as bit vectors if they contain more than one element. In general an array of memory words is needed to store the bits of one set (see Fig. 4). The membership test would be coded as follows:

```
Seti: ARRAY [0..k] OF BITSET;

e ∈ Seti   ≡   (e MOD 32) IN Seti [e DIV 32]
```

On a MC 68020 processor this produces the following six machine instructions including two divide instructions:

```
movl    _e:l,d1
divsll  #0x20,d2:d1
movl    _e:l,d3
divsl   #0x20,d3
movl    (_Set:l,d3:w:4),d3
btst    d2,d3
```

It is much more advantageous to store the sets *vertically* instead of *horizontally* as above. 32 sets can be stored side by side in a sufficient number of words (see Fig. 5). The membership test is coded as follows:

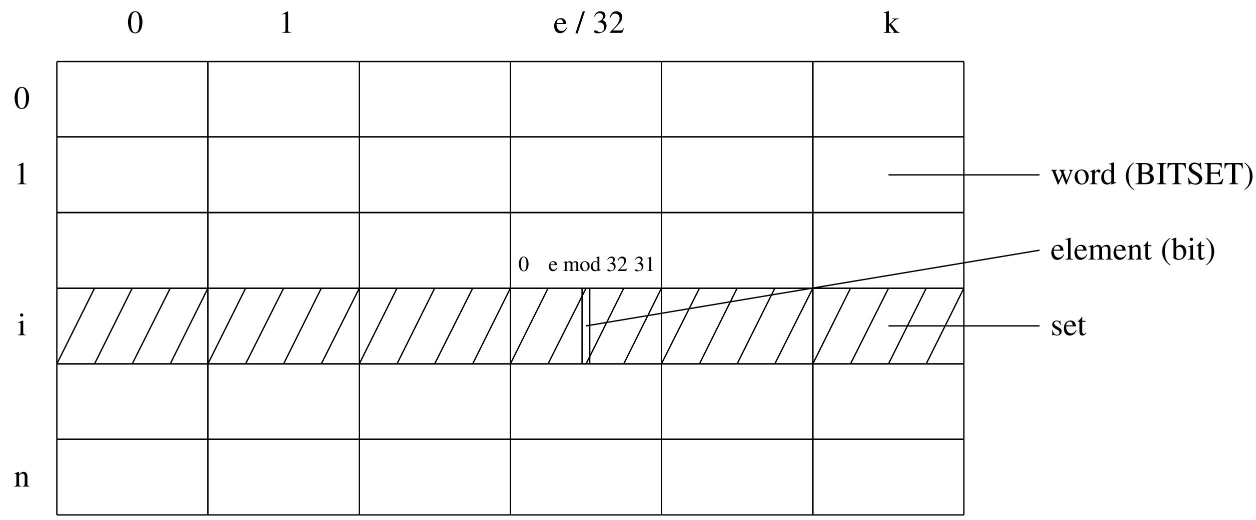

Fig. 4: horizontal set

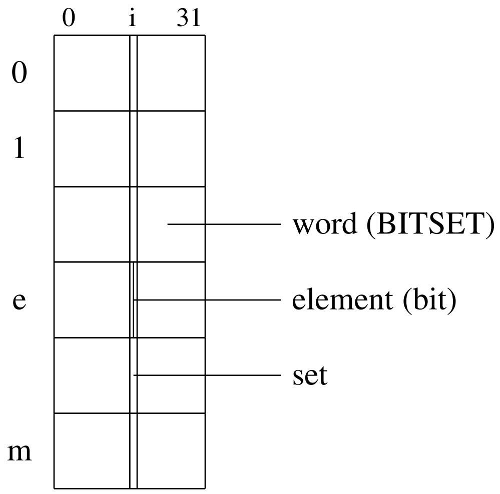

Fig. 5: vertical set

```
Set: ARRAY [0..m] OF BITSET;
e ∈ Seti    ≡    i IN Set [e]
```

Note, that i is a constant. Now on a MC 68020 processor two machine instructions suffice:

```
movw    _e+2:l,d4
btst    #4,(_Set:l,d4:w:4)
```

If the global variables e and Set could be stored in registers, the membership test could be done even with one machine instruction, only:

```
btst    #4,a0@(0,d0:l)
```

This implementation of the membership test has been previously described in [Gra88b]. In our case, the two ways of the membership test account for a difference of another 30% in the over all run time of the parser.

7. Summary

We presented the interesting features of the recursive descent parser generator *Ell* from the user's point of view and from the implementation aspects. The outstanding features of *Ell* are the L-attribution mechanism, its ability to handle non LL(1) grammars, and its automatic and comfortable error handling, which includes error reporting, recovery, and repair.

The generated parsers are extremely efficient in terms of run time. For example a Modula-2 parser runs at a speed of 55,000 tokens per second or 900,000 lines per minute on a MC 68020 processor (excluding scanning). The size of the parser is 25 K bytes and the run time of the generator is 10 seconds.

Acknowledgements

Whereas the author designed the generated code and the error recovery, the generator *Ell* was programmed by D. Kuske. The implementation of the global recovery sets was discovered independently by D. Schwartz-Hertzner and D. Kuske. B. Vielsack added the generation of C code, the L-attribution mechanism, and the disambiguating rules for non LL(1) grammars. The implementation of the set membership test is due to W. M. Waite and B. Gray.

References

[DuW81] R. C. Dunn and W. M. Waite, *SYNPUT*, Department of Electrical Engineering, Univ. of Colorado, Boulder, CO, Feb. 1981.

[Gra88a] R. W. Gray, Automatic Error Recovery in a Fast Parser, *Summer USENIX Conference*, , 1988.

[Gra88b] R. W. Gray, *γ-GLA - A Generator for Lexical Analyzers That Programmers can Use*, University of Colorado, Boulder, 1988.

[Gro88] J. Grosch, Generators for High-Speed Front-Ends, *LNCS 371*, (Oct. 1988), 81-92, Springer Verlag.

[GrV88] J. Grosch and B. Vielsack, The Parser Generators Lalr and Ell, Compiler Generation Report No. 8, GMD Forschungsstelle an der Universität Karlsruhe, Apr. 1988.

[Iro63] E. T. Irons, An Error Correcting Parse Algorithm, *Comm. ACM 6*, 11 (Nov. 1963), 669-673.

[M90] H. Mössenböck, Coco/R - A Generator for Fast Compiler Front-Ends, Report No. 127, Departement Informatik, ETH Zürich, Feb. 1990.

[ReM85] P. Rechenberg and H. Mössenböck, *Ein Compiler-Generator für Mikrocomputer*, Hanser, München, 1985.

[SMM84] M. Spenke, H. Mühlenbein, M. Mevenkamp, F. Mattern and C. Beilken, A Language Independent Error Recovery Method for LL(1) Parsers, *Software—Practice & Experience 14*, 11 (Nov. 1984), 1095-1107.

[Wil79] R. Wilhelm, Attributierte Grammatiken, *Informatik Spektrum 2*, 3 (1979), 123-130.

[Wir86] N. Wirth, *Compilerbau*, Teubner, Stuttgart, 1986.

Appendix 1: Scheme of the Code Generated for EBNF Constructs

```
                             (* Literal t                            *)
IF Token # t THEN RecoveryLiteral (t, Recover(t), GlobalrecoverySet);
ELSE Token := GetToken (); IsRepairMode := FALSE;
END;
                             (* Terminal t (with attribute ti)       *)
IF Token # t THEN RecoveryTerminal (t, Recover(t), GlobalrecoverySet, ti);
ELSE ti := Attribute; Token := GetToken (); IsRepairMode := FALSE;
END;
                             (* Nonterminal X (with attribute Xi)    *)
Union.LocalRecoverySet := Recover(X); X (SXSTEM.ADR (Union), Xi);

LOOP                         (* Optional part X = [ Y ]              *)
  IF Token ∈ FIRST (Y) THEN <Y> EXIT;
  ELSIF Token ∈ FOLLOW (X) OR IsRepairMode THEN EXIT; END;
  ErrorRecovery (Expected (X), Recover (X), GlobalRecoverySet);
END;

LOOP                         (* Iteration X = Y * or X = { Y }       *)
  IF Token ∈ FIRST (Y) THEN <Y>
  ELSIF Token ∈ FOLLOW (X) OR IsRepairMode THEN EXIT;
  ELSE ErrorRecovery (Expected (X), Recover (X), GlobalRecoverySet);
  END;
END;

LOOP                         (* Iteration X = Y + or X = Y { Y }     *)
  <Y>
  IF Token ∉ FIRST (Y) THEN
    IF Token ∈ FOLLOW (X) THEN EXIT; END;
    ErrorRecovery (Expected (X), Recover (X), GlobalRecoverySet);
    IF Token ∉ FIRST (Y) THEN EXIT; END;
  END;
END;

LOOP                         (* Iteration X = Y || Z or X = Y { Z Y } *)
  <Y>
  IF Token ∉ FIRST (Z) THEN
    IF Token ∈ FOLLOW (X) THEN EXIT; END;
    ErrorRecovery (Expected (X), Recover (X), GlobalRecoverySet);
    IF Token ∉ (FIRST (Y) ∪ FIRST (Z)) THEN EXIT; END;
  END;
  <Z>
END;

LOOP                         (* Alternative = X = Y1 | ... | Yn      *)
  CASE Token OF
  | FIRST (Y1) & FOLLOW (Y1): <Y1> EXIT;
      ...
  | FIRST (Yn) & FOLLOW (Yn): <Yn> EXIT;
  ELSE          (* Yd (1 <= d <= n): default alternative for error case *)
                (*                 duplication of one of the above    *)
    IF IsRepairMode THEN <Yd> EXIT; END;
    ErrorRecovery (Expected (X), Recover (X), GlobalRecoverySet);
  END;
END;
```

Appendix 2: Procedures for Error Recovery

```
TYPE tUnionPtr  = POINTER TO tUnion;
TYPE tUnion     = RECORD                    (* type for list elements      *)
                    GlobalRecoverySet   : tUnionPtr;
                    LocalRecoverySet    : SHORTCARD;
                  END;
TYPE tSet       = ARRAY [0..Upb] OF BITSET; (* type for bit sets           *)

VAR SetMemory   : ARRAY [0..169] OF tSet;   (* storage for horizontal sets *)

                                            (* test for set membership     *)
PROCEDURE IsElement (Set: tSet; Element: SHORTCARD): BOOLEAN;
  BEGIN
    RETURN Element MOD BitsPerBitset IN Set [Element DIV BitsPerBitset];
  END IsElement;
                            (* compute global recovery set and skip tokens *)
PROCEDURE SkipTokens (LocalRecoverySet: SHORTCARD; GlobalRecoverySet: tUnionPtr);
  VAR RecoverySet: tSet; i: SHORTCARD;
  BEGIN
    RecoverySet := SetMemory [LocalRecoverySet];
    INCL (RecoverySet [0], sEof);
    WHILE GlobalRecoverySet # NIL DO
      FOR i := 0 TO Upb DO RecoverySet [i] :=
        RecoverySet [i] + SetMemory [GlobalRecoverySet^.LocalRecoverySet] [i];
      END;
      GlobalRecoverySet := GlobalRecoverySet^.GlobalRecoverySet;
    END;
    WHILE NOT IsElement (RecoverySet, Token) DO
      Token := GetToken ();
    END;
    ErrorMessage (RestartPoint, Information, Line, Column);
    IsRepairMode := TRUE;
  END SkipTokens;

PROCEDURE ErrorRecovery (ExpectedSet      : SHORTCARD;
                         LocalRecoverySet : SHORTCARD;
                         GlobalRecoverySet: tUnionPtr);
  BEGIN
    IF NOT IsRepairMode THEN
      INC (ErrorCount);
      ErrorMessage (SyntaxError, Error, Line, Column);
      ErrorMessageI (ExpectedSymbols, Information, Line, Column, TokenSet,
                    SYSTEM.ADR (SetMemory [ExpectedSet]));
      SkipTokens (LocalRecoverySet, GlobalRecoverySet);
    END;
  END ErrorRecovery;
```

```
PROCEDURE RecoveryTerminal (Expected         : SHORTCARD;
                            LocalRecoverySet : SHORTCARD;
                            GlobalRecoverySet: tUnionPtr;
               VAR RepairAttribute: tScanAttribute); (* for terminals only *)
  BEGIN
    IF NOT IsRepairMode THEN
      INC (ErrorCount);
      ErrorMessage (SyntaxError, Error, Line, Column);
      ErrorMessageI (ExpectedSymbols, Information, Line, Column, Symbol,
                     SYSTEM.ADR (Expected));
      SkipTokens (LocalRecoverySet, GlobalRecoverySet);
    END;
    IF Token # Expected THEN
      ErrorMessageI (SymbolInserted, Repair, Line, Column, Symbol,
                     SYSTEM.ADR (Expected));
      ErrorAttribute (Expected, RepairAttribute);    (* for terminals only *)
    ELSE
      RepairAttribute := Attribute;                  (* for terminals only *)
      IF Token # sEof THEN Token := GetToken (); END;
      IsRepairMode := FALSE;
    END;
  END RecoveryTerminal;

PROCEDURE ErrorRecoveryLiteral (Expected         : SHORTCARD;
                                LocalRecoverySet : SHORTCARD;
                                GlobalRecoverySet: tUnionPtr);

   (* like ErrorRecoveryTerminal except as marked above *)
```

Appendix 3: Attribute Grammar to Compute the Recovery Sets

N	nonterminal
E	expression
t	terminal or literal
R	recovery set
In, Out	temporary attributes

```
N = E           {E.In := Ø;                                                     }
E = t           {E.R := FIRST (t) ∪ E.In; E.Out := E.R;                         }
E = N           {E.R := E.In; E.Out := FIRST (N) ∪ E.In;                        }
E = [ E1 ]      {E1.In := E.In; E.R := FIRST (E1) ∪ E.In; E.Out := E.R;         }
E = E1 *        {E1.In := E.In; E.R := FIRST (E1) ∪ E.In; E.Out := E.R;         }
E = E1 +        {E1.In := E.In; E.R := FIRST (E1) ∪ E.In; E.Out := E.R;         }
E = E1 || E2    {E1.In := FIRST (E2) ∪ E.In; E2.In := FIRST (E1) ∪ E.In;
                 E.R := FIRST (E1) ∪ FIRST (E2) ∪ E.In; E.Out := E.R;           }
E = E1 |...| En {E1.In := E.In; ... En.In := E.In;
                 E.R := FIRST (E1) ∪ ... ∪ FIRST (En) ∪ E.In; E.Out := E.R;   }
E = E1 E2       {E2.In := E.In; E1.In := E2.Out; E.R := E1.Out; E.Out := E.R;   }
```

Appendix 4: Example of Error Recovery

Source Program:

```
 1   MODULE Error;
 2   CONST M := 10, N = 100  X = 10;
 3   VAR , a, b, c;
 4
 5   PROCEDURE P;
 6   BEGIN
 7     s := 0; a = 5 * (b - 1 END;
 8
 9   BEGIN
10     > a > b;
11     WHILE a DO
12       BEGIN > b; - c := 0;
13       WHILE a > 0 BEGIN
14         IF ODD a c := c * - b;
15         b := 2 * b a := a /2
16       END;
17       P := 0; P; 666;
18     END .
```

Error Messages:

```
2, 9:   Error       syntax error
2, 9:   Information expected symbols: ’=’
2, 12:  Information restart point
2, 12:  Repair      symbol inserted : ’=’
2, 14:  Error       syntax error
2, 14:  Information expected symbols: ’;’
2, 16:  Information restart point
2, 16:  Repair      symbol inserted : ’;’
2, 25:  Error       syntax error
2, 25:  Information restart point
2, 25:  Repair      symbol inserted : ’;’
3, 5:   Error       syntax error
3, 7:   Information restart point
3, 14:  Error       syntax error
3, 14:  Information expected symbols: ’:’
3, 14:  Information restart point
3, 14:  Repair      symbol inserted : ’:’
3, 14:  Repair      symbol inserted : Ident
7, 13:  Error       syntax error
7, 13:  Information expected symbols: ’(’ ’:=’ ’;’
7, 19:  Information restart point
7, 26:  Error       syntax error
7, 26:  Information restart point
7, 26:  Repair      symbol inserted : ’)’
7, 29:  Error       syntax error
7, 29:  Information expected symbols: Ident
7, 29:  Information restart point
7, 29:  Repair      symbol inserted : Ident
10, 3:  Error       syntax error
10, 3:  Information expected symbols: Ident ’;’ ’CASE’ ’EXIT’ ’FOR’ ’IF’ ’LOOP’
                                      ’REPEAT’ ’RETURN’ ’WHILE’ ’WITH’
10, 5:  Information restart point
10, 7:  Error       syntax error
10, 7:  Information expected symbols: ’(’ ’:=’ ’;’
10, 9:  Information restart point
```

```
10, 9:  Repair      symbol inserted : ';'
12, 5:  Error       syntax error
12, 5:  Information expected symbols: Ident ';' 'CASE' 'EXIT' 'FOR' 'IF' 'LOOP'
                                      'REPEAT' 'RETURN' 'WHILE' 'WITH'
12, 13: Information restart point
12, 16: Error       syntax error
12, 16: Information expected symbols: Ident ';' 'CASE' 'EXIT' 'FOR' 'IF' 'LOOP'
                                      'REPEAT' 'RETURN' 'WHILE' 'WITH'
12, 18: Information restart point
13, 17: Error       syntax error
14, 7:  Information restart point
14, 7:  Repair      symbol inserted : 'DO'
14, 14: Error       syntax error
14, 14: Information restart point
14, 14: Repair      symbol inserted : 'THEN'
14, 16: Error       syntax error
14, 16: Information restart point
14, 16: Repair      symbol inserted : ';'
14, 25: Error       syntax error
14, 25: Information expected symbols: Ident Integer Real String '(' '{' 'NOT'
14, 25: Information restart point
14, 25: Repair      symbol inserted : Integer
15, 18: Error       syntax error
15, 18: Information restart point
15, 18: Repair      symbol inserted : ';'
17, 16: Error       syntax error
17, 16: Information expected symbols: Ident ';' 'CASE' 'EXIT' 'FOR' 'IF' 'LOOP'
                                      'REPEAT' 'RETURN' 'WHILE' 'WITH'
17, 19: Information restart point
18, 7:  Error       syntax error
18, 7:  Information restart point
18, 7:  Repair      symbol inserted : 'END'
18, 7:  Repair      symbol inserted : 'END'
18, 7:  Repair      symbol inserted : Ident
```

Contents

	Abstract	1
1.	Introduction	1
2.	L-Attribution	1
2.1.	User's View	1
2.2.	Implementation	2
3.	Non LL(1) Grammars	2
4.	Error Recovery	3
4.1.	User's View	3
4.2.	Implementation	4
4.2.1.	Expected Symbols	4
4.2.2.	Recovery Sets	4
4.2.3.	Error Repair	6
5.	Related Work	7
6.	Implementation Issues	7
7.	Summary	9
	Acknowledgements	9
	References	10
	Appendix 1: Scheme of the Code Generated for EBNF Constructs	11
	Appendix 2: Procedures for Error Recovery	12
	Appendix 3: Attribute Grammar to Compute the Recovery Sets	13
	Appendix 4: Example of Error Recovery	14