Deterministic Syntax Error Recovery

Gordon V. Cormack
Department of Computer Science

Research Report CS-87-39
June 1987

Deterministic Syntax Error Recovery

Gordon V. Cormack

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

EDU, CDN, CSNET: gvcormack@waterloo
UUCP, BITNET: gvcormac@water

1. Introduction

In many applications, the behaviour of a parser on invalid input is as important as its ability to parse correct input. It is desirable that the parser scan the entire input and report all syntax errors. It is further desirable that the parser identify only errors – no spurious reports should be generated.

LR(1) parsers [cf. Aho 86] have the desirable property that they detect the *first* syntax error as early as possible in a left-to-right scan of the input. Specifically, if the set of valid input strings is the language L and the invalid input is x, an LR(1) parser finds a string w and a symbol a such that $way = x$ and $\exists z\ wz \in L$ and $\nexists z\ waz \in L$. Upon detecting the first syntax error, the LR(1) parser halts, leaving y (the remainder of the input x) unchecked for errors. In order to report *all* the syntax errors in x, the parse must be resumed to recognize y. One approach to resuming the parse is to substitute a valid prefix v in place of wa in the input. The parser proceeds to recognize vy. It is difficult to make an appropriate choice for v: it is possible that $vy \notin L$ when another choice v' would result in $v'y \in L$. In this case, a spurious error report occurs.

The suffix parser described here recognizes y if and only if $\exists v\ vy \in L$, and therefore circumvents the problem of finding a suitable v. If y is invalid, the suffix parser identifies the next syntax error: it finds the string w' and the symbol a' such that $w'a'y' = y$ and $\exists v,z\ vw'z \in L$ and $\nexists v,z\ vw'a'z \in L$. Repeated application of the suffix parser yields a *suffix analysis* of x: a set of strings w_k and symbols a_k such that

$x = w_0a_0w_1a_1\ldots\ \ldots w_{n-1}a_{n-1}w_n$
$\exists z\ w_0z \in L$
$\nexists z\ w_0a_0z \in L$
$\exists v,z\ vw_kz \in L\ \{0<k<n\}$
$\nexists v,z\ vw_ka_kz \in L\ \{0<k<n\}$
$\exists v\ vw_n \in L$.

In *Noncorrecting syntax error recovery,* [Ric 85] Richter examines suffix analysis (and also *interval analysis*, which requires a right-to-left version of the recognizer presented here) and argues its desirability in reporting syntax errors. Richter's work provides the primary motivation for the suffix parser described here, and the reader is directed to that work for further elaboration. Hammond and Rayward-Smith have published a general survey on syntax error handling. [Ham 84]

The suffix parser exactly implements noncorrecting error recovery, yet is generated automatically, and runs with the same efficiency as an LR parser. In this paper we give two constructions for suffix parsers, namely (1)SLR(1) and (1)LR(1). If L is the language generated by a context free grammar G (e.g. the grammar for a programming language), a suffix parser recognizes the language $FIN = \{x \mid \exists w\ wx \in L\}$. The constructions take as input the grammar G, and produce as output the action table for a single-stack shift-reduce parser for *FIN*. For a subset of possible context-free grammars, the action table is conflict-free and hence the suffix parser is deterministic. The (1)LR(1) construction yields a deterministic suffix parser if G is 1–1 *bounded context* [Flo 64], while the (1)SLR(1) parser is deterministic for a smaller class of grammars, which we call 1–1 *simple bounded context*. Both these classes of grammars are symmetric: if G is a member of the class, so is *REV*(G) (the grammar generating *REV*(L) formed by reversing all productions in G). Therefore, right-to-left prefix parsers are easily generated from the same grammars.

2. Related work

A direct approach to constructing these recognizers is to define a grammar G' such that $L' = FIN$. G' can be constructed automatically, but the construction does not in general maintain the LR (or LL) property of G. It is undecidable whether $G' \in$ LR can be constructed for $G \in$ LR. [Ric 85] If a suitable G' can be constructed, it is unlikely to preserve the phrase structure of G (such as precedence and associativity of operators), and it is very difficult to verify that L' is in fact equal to *FIN*. In order to do translation as well as error recovery, it would be necessary for the compiler to contain two parsers: one for G and another for G'. The construction of the reverse recovery parser would require yet another grammar G'' such that $L'' =$ REV(*FIN*). Harrington has experimented with writing these LR suffix grammars for a subset of Pascal and concluded that the technique is impractical for real programming languages [Har 86]. Another possible approach is to discard the requirement that G' be LR (or even unambiguous) and to use the general context-free parsing algorithm due to Earley [Ear 70]. This approach to error recovery is inefficient (cubic in the length of input) and suffers from the general shortcomings mentioned in this paragraph.

Another approach is to adapt a parser for G to recognize *FIN* instead of L. Graham and Rhodes [Gra 75] use a precedence parser to perform *right condensation* – parsing beyond the point of error. This right condensation performs a correct parse if the remaining input is valid, but does not have the correct prefix property, and does not necessarily detect further errors at all. Instead, right condensation is used as a tool to gather more information in order to repair

the original error. Pennello and DeRemer have developed a *forward move algorithm* (FMA) [Pen 79] which is the LR analogue of right condensation. FMA starts by assuming that the parse might be in *any* LR state and manipulates sets of possibly valid states. So long as each LR state agrees on the action to be taken with the upcoming symbol, the FMA parse continues. The algorithm terminates when an error is detected or when there are conflicting actions among the valid states. The stated purpose of FMA is to provide information to a repair strategy, but it is a significant step towards a recovery parser: any fragment accepted by FMA is in *FIN*. Bumbulis [Bum 86] gives a construction that transforms any pushdown automaton (PDA) that recognizes L into a new PDA that recognizes *FIN*. Unfortunately, the new PDA may be nondeterministic. It may be made deterministic by enlarging the language recognized to $FIN' \supset FIN$. While this transformation causes the PDA to miss some fraction of the possible errors, Bumbulis claims that in practice this fraction is not large.

Richter [Ric 85] mentions a method for constructing a deterministic recognizer for *FIN* directly from G – the use of the construction resembles the use of an LR parser generator. The construction succeeds for a subclass of context-free grammars that is not precisely defined by Richter. In order to be amenable to this construction method, G must be transformed with considerable skill. Furthermore, the transformation often destroys the semantic structure of G (such as the priority and associativity of operators), implying that separate grammars (and therefore parsers) are necessary for semantic analysis and error recovery (yet another grammar is necessary to perform interval analysis). From his work with the construction algorithm, Richter conjectures that the bounded-context property of G is sufficient for the existence of a deterministic recognizer for *FIN*. The (1)LR(1) construction shows the conjecture to be true.

3. Notation and definitions

We wish to construct a suffix parser for a context-free grammar $G = (N,T,R,S)$, where N is a set of nonterminal symbols, T is a set of terminal symbols, R is a set of rules of the form $A \rightarrow \alpha$, and $S \in N$ is the start symbol.

The following conventions are used: the letters A–C and S denote nonterminal symbols; a–d denote terminal symbols; W–Z denote terminal or nonterminal symbols; u–z denote strings of terminal symbols; and α–δ denote strings of terminal and nonterminal symbols. We assume that the only rule defining S is of the form $S \rightarrow \vdash A \dashv$, where $\vdash$ and $\dashv$ are special end markers that do not appear elsewhere in R. We assume further that G is cycle-free and that all nonterminal symbols are useful.

$\alpha A \beta \Rightarrow \alpha\gamma\beta$ means $A \rightarrow \gamma \in R$.

$\alpha A x \underset{rm}{\Rightarrow} \alpha\gamma x$ means $A \rightarrow \gamma \in R$.

$\alpha \Rightarrow^* \beta$ means $\alpha = \beta$ or $\exists\gamma\ \alpha \Rightarrow \gamma \Rightarrow^* \beta$.

$\alpha \Rightarrow^+ \beta$ means $\exists\gamma\ \alpha \Rightarrow \gamma \Rightarrow^* \beta$.

For a set Q, $Q \Rightarrow \beta$ means $\exists \alpha \in Q \ \alpha \Rightarrow \beta$

The language generated by G is $L = \{x \mid S \Rightarrow^* x\}$.

The suffix language of L is $FIN = \{x \mid \exists w \ wx \in L\}$.

Definition. The set of sentential suffixes is
$SUFF = \{\beta \mid \exists \alpha \ S \Rightarrow^* \alpha\beta\}$.

Definition. The set of 1-reducible sentential suffixes is
1-RED $= \{a\beta \mid \exists \alpha \ SUFF \Rightarrow^* a\alpha \Rightarrow^+ a\beta\}$.

Definition. The set of 1-irreducible sentential suffixes is
1-IRR $= \{a\beta \in SUFF - 1\text{-}RED\}$.

Definition. A rightmost 1-suffix derivation of $x \in FIN$ is
$\beta_0 \underset{rm}{\Rightarrow} \beta_1 \underset{rm}{\Rightarrow} \ldots \ldots \underset{rm}{\Rightarrow} \beta_n$ where $\beta_0 \in 1\text{-}IRR$ and $\beta_n = x$.

4. The suffix parser

The suffix analysis algorithm is identical whether the (1)LR(1) or the (1)SLR(1) construction is used. The algorithm uses the tables *SHIFT*, *REDUCE*, and *GOTO* that are constructed from G. Using a run-time stack, the suffix parser repeatedly finds the leftmost reducible phrase in the input, and thus constructs a rightmost 1-suffix derivation in reverse. For input $\vdash x \dashv$, the algorithm is as follows.

```
push start_state on stack
for every token t in ⊢x⊣ do
   while REDUCE(top_of_stack, t) = A→α do
      pop |α| elements from stack
      push GOTO(top_of_stack, A) on stack
   if SHIFT(top_of_stack, t) = n then
      push n on stack
   else
      report error at t
      push start_state on stack
```

5. The (1)SLR(1) construction.

The first step in constructing the (1)SLR(1) parser is to construct the (1)LR(0) parser. The (1)LR(0) parser is based on a finite state automaton that recognizes the leftmost reducible phrase. It is constructed first as a nondeterministic automaton M whose states are called (1)LR(0) *items*. Then the subset construction is applied to form a deterministic automaton D whose states and transitions are the (1)LR(0) parser. Finally, lookahead sets are computed to construct the (1)SLR(1) parser.

The set of (1)LR(0) items is the union of
the initial state: $\{START\}$
LR(0) items: $\{A \rightarrow \alpha . \beta \mid A \rightarrow \alpha\beta \in R\}$
suffix(0) items: $\{A \rightarrow \cdots \beta \mid \exists \alpha\ A \rightarrow \alpha\beta \in R\}$.

The nondeterministic transitions $\Delta(s,t)$ from state s on symbol t are defined by the following rules:

$\Delta(START,a) = \{A \rightarrow \cdots \beta \mid \exists \alpha\ A \rightarrow \alpha a \beta \in R\}$
$\Delta(A \rightarrow \cdots X\alpha, X) = \{A \rightarrow \cdots \alpha\}$
$\Delta(A \rightarrow \cdots B\alpha, \epsilon) = \{B \rightarrow . \beta \mid B \rightarrow \beta \in R\}$
$\Delta(A \rightarrow \cdots, \epsilon) = \{B \rightarrow \cdots \beta \mid \exists \alpha\ B \rightarrow \alpha A \beta \in R\}$
$\Delta(A \rightarrow \alpha . X\beta, X) = \{A \rightarrow \alpha X . \beta\}$
$\Delta(A \rightarrow \alpha . B\beta, \epsilon) = \{B \rightarrow . \beta \mid B \rightarrow \beta \in R\}$.

The equivalent deterministic automaton is achieved using the standard subset construction. Each state d in D is a subset of the states of M. The initial state is defined to be

$start\_state = \epsilon\text{–}closure(\{START\})$.

The (1)SLR(1) parser uses the following action tables, which define D.

$SHIFT(d,a) = \{\Delta(i,a) \mid i \in d\}$
$GOTO(d,A) = \{\Delta(i,A) \mid i \in d\}$
$REDUCE(d,a) = A \rightarrow \alpha \mid A \rightarrow \alpha . \in d$ and $a \in FOLLOW(A)$, where
$FOLLOW(A) = \{a \mid \exists \gamma, \delta\ S \Rightarrow^* \gamma A a \delta\}$.

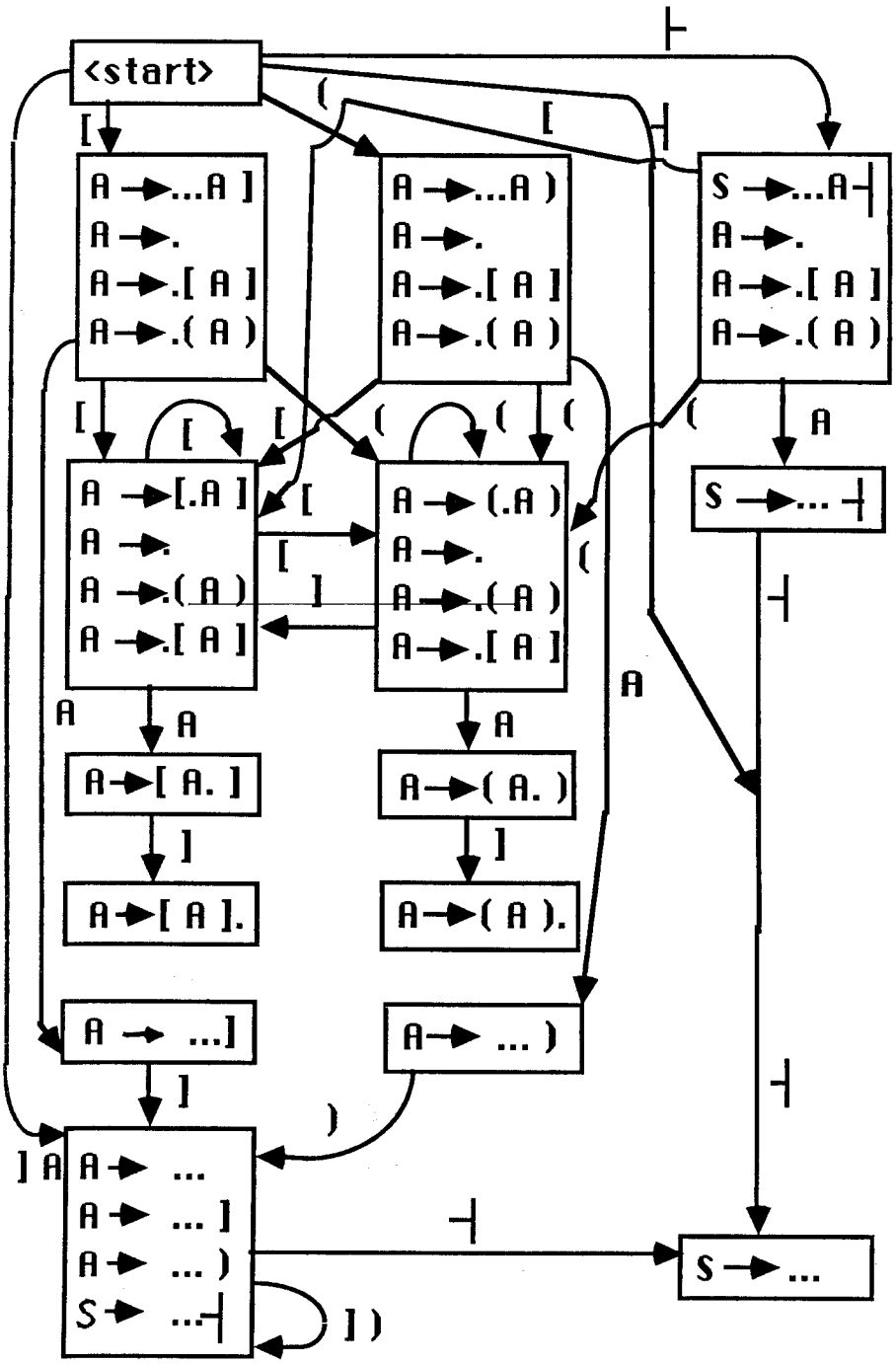
G is (1)SLR(1) if, for any d and a, at most one of $SHIFT(d,a)$ and $REDUCE(d,a)$ is defined.

6. An example

As a simple example, we use the following grammar that generates all strings of nested pairs of brackets and parentheses.

S → ⊢ A ⊣
A → ϵ
A → [A]
A → (A)

The suffix parser recognizes complete strings such as ⊢(([()])) ⊣ as well as suffixes like [()]))))))]]]⊣ or]]]))) ⊣ but not (]]]] or [[(()))]⊣. The (1)LR(0) machine is as follows, and the (1)SLR(1) actions are easily derived from it (the follow set for each reduce item is { ⊣,),] }).


<start>
A →...A]
A →.
A →.[A]
A →.(A)
A →...A)
A →.
A →.[A]
A →.(A)
S →...A⊣
A →.
A →.[A]
A →.(A)
A →[.A]
A →.
A →.(A)
A →.[A]
A →(.A)
A →.
A →.(A)
A →.[A]
S →... ⊣
A→[A.]
A→(A.)
A→[A].
A→(A).
A → ...]
A→ ...)
A → ...
A → ...]
A → ...)
S → ...⊣
]]
S →...

7. The (1)LR(1) parser

The (1)LR(1) parser is constructed in the same manner as the (1)SLR(1) parser, and the parsing algorithm is identical. The difference between (1)LR(1) and (1)SLR(1) is in the states of M and in the calculation of *SHIFT*, *GOTO*, and *REDUCE*.

The states of M are the (1)LR(1) items:

Definition. The set of (1)LR(1) items is the union of
the initial state: $\{START\}$
LR(1) items: $\{A \rightarrow \alpha . \beta : a \mid A \rightarrow \alpha\beta \in R\}$
suffix(1) items: $\{A \rightarrow \cdots \beta : a \mid \exists \alpha\ A \rightarrow \alpha\beta \in R\}$.

The transition function Δ is defined by

$\Delta(\text{START,a}) = \{\text{A}\rightarrow ...\beta : \text{b} \mid \text{A}\rightarrow \alpha \text{a} \beta \in R \text{ and b} \in \text{FOLLOW(A)}\}$
$\Delta(\text{A}\rightarrow ...\text{X}\alpha : \text{a,X}) = \{\text{A}\rightarrow ...\alpha : \text{a}\}$
$\Delta(\text{A}\rightarrow ...\text{B}\alpha : \text{a}, \epsilon) = \{\text{B}\rightarrow .\beta : \text{a} \mid \text{B}\rightarrow \beta \in R \text{ and } \alpha \Rightarrow^* \epsilon\} \cup \{\text{B}\rightarrow .\beta : \text{b} \mid \text{B}\rightarrow \beta \in R \text{ and } \alpha \Rightarrow^* \text{b}\gamma\}$
$\Delta(\text{A}\rightarrow ... : \text{a},\epsilon) = \{\text{B}\rightarrow ...\beta : \text{b} \mid \text{B}\rightarrow \alpha\text{A}\beta \in R \text{ and b} \in \text{FOLLOW(B)}\}$
$\Delta(\text{A}\rightarrow \alpha.\text{X}\beta : \text{a,X}) = \{\text{A}\rightarrow \alpha\text{X}.\beta : \text{a}\}$
$\Delta(\text{A}\rightarrow \alpha.\text{B}\gamma : \text{a},\epsilon) = \{\text{B}\rightarrow .\beta : \text{a} \mid \text{B}\rightarrow \beta \in R \text{ and } \gamma \Rightarrow^* \epsilon\} \cup \{\text{B}\rightarrow .\beta : \text{b} \mid \text{B}\rightarrow \beta \in R \text{ and } \gamma \Rightarrow^* \text{b}\delta\}$

The (1)LR(1) parser has the same state set as D, the deterministic equivalent of M. The actions are:

$\text{SHIFT}(d, \text{a}) = \{\Delta(i,a) \mid i \in d\}$
$\text{GOTO}(d, \text{A}) = \{\Delta(i,A) \mid i \in d\}$
$\text{REDUCE}(d, \text{a}) = \text{A}\rightarrow \alpha \mid \text{A}\rightarrow \alpha. : \text{a} \in d$.

G is (1)LR(1) if at most one *SHIFT* or *REDUCE* is defined for any d and a.

8. A recovery parser for Pascal

To test the efficacy of the (1)SLR(1) technique, the parser generator was written (using Modula 2) and an amenable Pascal grammar was written. The grammar contains 274 productions and the resulting parser has 1000 states. An SLR(1) grammar written by the author has 156 productions and resulted in an SLR machine with 400 states. While the (1)SLR(1) versions are significantly larger, we still consider their size to be reasonable. The semantic structure of Pascal is largely preserved in the (1)SLR(1) grammar, but some transformations were required to resolve conflicts.

Although empty and unit rules in the grammar do not automatically cause conflicts in the (1)SLR(1) parser (witness the example grammar in section 6), they were often involved in conflicts. Therefore, the grammar was written to avoid the use of null rules, and the parser generator was modified to eliminate unit productions during the construction process. Now that we have gained some experience with the parser generator, we plan to re-introduce the empty rules.

A number of features of Pascal proved troublesome. In general, conflicts arise when distinct clauses of the grammar generate program fragments that could be confused with one another, but are not identical. For example, the list "red,blue,green" might be an actual parameter list, a list of variables in a declaration, or a list of case labels. Also, the list "10,-x,30" might be an actual parameter list or a list of case labels. If it is a parameter list, "-x" must be parsed as an expression, but if it is a list of case labels, it must be parsed as a constant. The general approach we used to resolve these cases was to write a separate subgrammar to describe the intersections between the troublesome constructs. The following example illustrates this process using a simplified grammar for Pascal-like expressions and lists. The following grammar is SLR(1) but not (1)LR(1) because the nonterminals "consts", "exprs", and "ids" may be confused following a "(".

s → ⊢ clause ⊣
clause → CONSTS (consts) | EXPRS (exprs) | IDS (ids)
consts → const | consts , const
exprs → expr | exprs , expr
ids → id | ids , id
const → id | + id
expr → sexpr | sexpr = sexpr
sexpr → term | + term
term → factor | term * factor
factor → id | (expr)

The next grammar generates the same language, but is (1)SLR(1).

s → ⊢ clause ⊣
clause → CONSTS (consts) | EXPRS (exprs) | IDS (ids)
ids → id | ids , id
consts → ids | consts!ids
consts!ids → const!id | ids , const!id | consts!ids , const
exprs → consts | exprs!consts
exprs!consts → expr!const | consts , expr!const | exprs!consts , expr
const → id | const!id
const!id → + id
expr → const | expr!const
expr!const → sexpr!const | sexpr = sexpr
sexpr → const | sexpr!const
sexpr!const → term!id | + term!id
term → id | term!id
term!id → factor!id | term * factor
factor → id | factor!id
factor!id → (expr)

The transformation process can be tedious, but it is straightforward. We plan to automate it within the (1)SLR(1) generator. Note that the phrase structure of the grammar is preserved.

We have at the moment implemented only the (1)SLR(1) parser. Our conclusions from writing a recovery parser for Pascal is that the technique is a viable way to generate simple and effective syntax error recovery in parsers. We have made no attempt to perform semantic actions after detecting a syntax error, but because the recovery parser reduces any complete phrases within the remaining program, we feel that reasonable semantic recovery should be possible. We note also, that after a suffix analysis the parse stack contains a list of fully reduced sentential fragments. One could attempt error repair after the fact by attempting to fit these fragments together to form a valid sentential form. Other possible applications of the (1)LR(1) algorithm include the construction of incremental and parallel parsers.

References

[Aho 86]
Aho A., Sethi R. and Ullman J., *Compilers — Principles, Techniques, and Tools*, Addison-Wesley (1986).

[Bum 86]
Bumbulis P., *A practical non-correcting syntax error recovery algorithm*, M.Math thesis, University of Waterloo (1986).

[Ear 70]
Earley J., An efficient context-free parsing algorithm, *Commun. ACM* 13:2 (1970), 94-102.

[Flo 64]
Floyd R., Bounded context syntax analysis, *Commun. ACM* 7:2 (1964), 62-67.

[Gra 75]
Graham S. and Rhodes S., Practical syntactic error recovery, *Commun. ACM* 18:11, (1975), 639-650.

[Ham 84]
Hammond K. and Rayward-Smith V., A survey on syntactic error recovery and repair, *Computer Languages* 9:1 (1984), 51-67.

[Har 86]
Harrington J., *Error recovery using a suffix parser for Pascal*, M.Math essay, University of Waterloo (1986).

[Pen 79]
Pennello T. and DeRemer F., A forward move algorithm for LR error recovery, *5th ACM Symposium on Principles of Programming Languages*(1979), 241-254.

[Ric 85]
Richter H., Noncorrecting syntax error recovery, *ACM Trans. Programming Languages and Systems* 7:3 (1985), 478-489.