IMMEDIATE ERROR DETECTION IN STRONG LL(1) PARSERS

by

C. N. Fischer

K. C. Tai

D. R. Milton

Computer Sciences Technical Report #332

August 1978

Immediate Error Detection In Strong LL(1) Parsers

C. N. Fischer
University of Wisconsin - Madison
Madison, Wisconsin 53706

K. C. Tai†
Department of Computer Science
North Carolina State University
P.O. Box 5972
Raleigh, N.C. 27650

D. R. Milton
Bell Laboratories
Naperville, IL 60540

Keywords: Compilers, Parsing, LL(1), Syntactic Errors, Error Correction

---

†This research has been supported in part by the National Science Foundation Grant MCS 77-24582

Abstract

An algorithm is presented which guarantees that no erroneous actions are performed by a Strong LL(1) parser while parsing an incorrect input. The class of Strong LL(1) grammars for which this algorithm is usable appears to closely coincide with grammars actually used in practice. Further any Strong LL(1) grammar can be algorithmically transformed into a form suitable for use with this algorithm.

1. Introduction

One of the best known and most popular tabular parsing techniques is LL(k) [1]. LL(k) is very closely related to another parsing technique, Strong LL(k). In fact, the two techniques differ only in details of how parsing tables are created.

The difference between LL(k) and Strong LL(k) parsing is often blurred, in large measure because in the case of one symbol look-ahead the classes of context-free grammars parsable by the two techniques are identical [1]. Strong LL(1) parsers require smaller parsing tables and thus LL(1) parsers are almost never used in practice.

However LL(1) and Strong LL(1) parsers are not entirely equivalent: they differ slightly in how (and when) syntax errors are detected. Both have the correct prefix property; that is, symbols accepted by them can be guaranteed to form a prefix of some sentence in the language. LL(1) has the even stronger property that an error is detected as soon as an erroneous input symbol is first encountered. We shall call this property the immediate error detection property. As a result of this property, LL(1) parsers never perform any parsing action on an erroneous input symbol. On the other hand, Strong LL(1) parsers do not have the immediate error detection property [1, 5]. As a result, incorrect input symbols can sometimes induce a Strong LL(1) parser to perform incorrect parsing actions.

Such incorrect parsing actions can seriously interfere with syntactic error recovery or correction. Consider the following Strong

LL(1) grammar† presented in [4]:

G1: $S \rightarrow E\$$

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E' \mid \lambda$

$T \rightarrow a \mid (E)$

This grammar generates simple infix expressions in +; it is very similar in form to LL(1) grammars used in practice to specify arithmetic expressions. It also happens to be insert-correctable [4]; that is, all syntax errors in G1 can be corrected via suitable insertions to the immediate left of error symbols.

Now consider a Strong LL(1) parser for G1 processing an input of a)...$. The parsing stack sequence is $S \vdash E\$ \vdash TE'\$ \vdash aE'\$ \vdash E'\$ \vdash \$$. At this point the parser announces "error", but it is too late — there is no useful information left on the parsing stack to guide a correction (or recovery) algorithm. In fact, the only possibility left is to delete all remaining input — which is equivalent to simply terminating parsing.

The problem above occurred because ')' induced the incorrect parse move $E'\$ \vdash \$$ (because $')' \in \text{Follow}_1(E')$). Had we the immediate error detection property, parsing would have stopped with a parse stack of E'$ and a simple insertion of '+(a' would have effected a correction.

Naturally LL-based error-recovery and correction schemes demand the immediate error detection property. A number of ways of obtaining it have been suggested:

---

† λ is used to denote the empty or null string.

(1) In [4] an LL(1) rather than a Strong LL(1) parser is suggested. However, for one typical grammar this tripled the number of non-error entries in the parsing table.

(2) Ghezzi, in [5], suggests transforming Strong LL(1) grammars into structurally equivalent Strong LL(1) grammars which have the immediate error detection property. However, his construction is equivalent to that presented in [1] in building LL(1) parsers. Thus an increase in parsing table size comparable to that noted in (1) can occur.

(3) In [3] a method is presented of saving parser moves in a queue until they are known to be correct. This approach is unattractive in that it can significantly increase the time and space requirements of a parser. In the worst case this queuing technique can require $O(n)$ extra space and $O(n^2)$ extra time to process an input of size n[†] .

None of the above proposals is very attractive in that each incurs a rather substantial overhead in extra space and/or time requirements. We present a simple and efficient method for immediate error detection in a subset of Strong LL(1) grammars termed nonnullable. This subset is of interest in that it appears to closely coincide with those Strong LL(1) grammars actually used in practice. Further, it will be shown that any Strong LL(1) grammar can be algorithmically transformed into

---

[†]The $O(n^2)$ increase in time is especially onerous in that linear-time LL(1) error correctors are known [4] .

an equivalent nonnullable Strong LL(1) grammar.

2. Immediate Error Detection

Following Aho & Ullman [2], we term a context-free grammar nonnullable[†] if and only if each production is of the form $A \to \lambda$ or $A \to X_1 \ldots X_k$ $(k \geq 1)$ where $X_1 \ldots X_k \not\Rightarrow^+ \lambda$. In nonnullable grammars λ can only be derived directly (in one step). For our purposes, such grammars are of interest in that the range of possible moves that an error symbol can induce in a nonnullable Strong LL(1) parser is severely limited.

Lemma 2.1

In a Strong LL(1) parser for a nonnullable grammar, the only parser moves an error symbol can induce are predictions of productions of the form $A \to \lambda$.

Proof

Clearly, the only parser moves an error symbol can induce in any Strong LL(1) parser are predictions. By construction of Strong LL(1) parsers and the definition of nonnullable grammars, a non-λ-production $A \to X_1 \ldots X_k$ will be predicted for an input symbol a if and only if $a \in \text{First}_1(X_1 \ldots X_k)$; that is $X_1 \ldots X_k \Rightarrow^* a \ldots$. But such a prediction is obviously correct. □

Since only predictions of λ-productions can possibly be erroneous, we modify the standard Strong LL(1) parsing algorithm (Algorithm 2.2) to give λ-productions special scrutiny. In particular, function

---

[†]Actually Aho & Ullman require that $X_1 \not\Rightarrow^+ \lambda$. Our definition simplifies Algorithm 3.1.

CHECK_LAMBDA_PROD determines whether or not the prediction of the λ-production is correct. It does this by noting that the net effect of predicting a lambda production is to pop the parse stack. It therefore considers the parsing action which would be induced by the new stack top and the current input symbol. If this action is pop, accept, or the prediction of a non-λ-production, then the λ-production must be correct. If the action induced is error then the λ-production is obviously incorrect. If the parsing action is the prediction of another λ-production, then we iterate the above by considering the next symbol down in the parse stack (obviously this process must halt). We thus obtain:

Algorithm 2.2 - A parser for nonnullable Strong LL(1) grammars.

```
{ Assume X_m...X_0   (m ≥ 0, X_0 = $) is the parse stack with X_m the
     top element. Let a be the current input symbol.
     T is the parser action table }
LOOP {FOREVER}
  CASE T [X_m, a] OF
    POP:  Pop X_m from parse stack;
          read next input symbol;
    ACCEPT:  Terminate parse successfully;
    ERROR:   Invoke Error Corrector;
    PREDICT: {Assume prediction is  X_m → α }
        IF (α ≠ λ) OR
           (α = λ) AND  CHECK_LAMBDA_PROD
        THEN  Pop X_m from parse stack;
              Push α onto the parse stack
```

Algorithm 2.2 continued

```
                ELSE  Invoke Error Corrector FI
    END {CASE}
END {LOOP}                                                    □
```

Note that in the above algorithm we assume that CHECK_LAMBDA_PROD is only invoked if $\alpha = \lambda$; that is, OR is evaluated in an optimized, "short-circuit" mode.

Algorithm 2.3 - Check whether a predicted λ-production is correct.

```
FUNCTION  CHECK_LAMBDA_PROD : BOOLEAN ;
   { X_m...X_0  (m ≥ 0, X_0 = $) is the current parse stack;  a  is the
     current input symbol }
   FOR i := m-1  DOWNTO  0  DO
      CASE  T [X_i, a]  OF
       POP, ACCEPT : RETURN (TRUE) ;
       ERROR : RETURN (FALSE) ;
       PREDICT :  {Assume X_i → β  is predicted}
            IF  β ≠ λ  THEN RETURN (TRUE) FI
      END {CASE}
   END {FOR}
END {CHECK_LAMBDA_PROD}                                       □
```

Theorem 2.4

Any nonnullable Strong LL(1) grammar parsed with Algorithm 2.2 will have the immediate error detection property.

Proof

Follows from the above discussion. □

Algorithm 2.2 is a very attractive way of providing for immediate error detection in Strong LL(1) parsers. CHECK_LAMBDA_PROD is a very compact routine requiring essentially no additional run-time data structures. It is only called in those cases when an erroneous parser move might actually occur. Execution speed of the routine should be no problem. Normally only a very few parse stack entries will need to be examined. In the case of a bounded depth parse stack (which is invariably used in practice), the entire parse stack can be examined, if necessary, in constant time. Even if an unbounded depth parse stack is used, a linear time parser can be guaranteed. For each stack symbol visited by CHECK_LAMBDA_PROD with a current input symbol of a , we table whether or not a leads to an error move. If such a stack symbol is subsequently considered by CHECK_LAMBDA_PROD with an input of a , the information tabled tells us whether or not an error move will be found. Since each stack symbol needs to be processed only once (at worst) for each terminal, linearity can readily be established.

Algorithm 2.2 also interfaces well with syntax-directed compilers. Since no incorrect parser moves can ever occur, we need not worry about "undoing" semantic actions initiated by such moves. This is especially important for even if incorrect parse moves could somehow be reversed, semantic actions (such as code generation or symbol table manipulation)

must often be considered irrevocable.

Some error recovery schemes, for simplicity's sake, discontinue semantic checking and code generation after a syntax error is discovered. They merely seek to continue syntax checking. For these schemes we can improve overall efficiency by no longer checking in advance (in Algorithm 2.2) whether or not a prediction of a λ-production is valid. Rather, we wait until an error is detected by the parser and then use the results of Lemma 2.1 to "undo" any illegal predictions of λ-productions.

For nonnullable Strong LL(1) grammars, the sequence of parser moves preceding detection of an error is quite constrained: a terminal symbol is popped† and then 0 or more (incorrect) λ-productions are predicted. Since parse stacks are invariably implemented in arrays, symbols "popped" are not lost until they are overwritten by subsequent pushes. Thus, in the above case it is trivial to "undo" any illegal predictions of λ-productions: we simply "move up" the top of stack pointer to a position just below the first terminal symbol encountered. At this point, the stack is restored to the state is was in just after the last (correct) input symbol was popped. This approach is attractive in that correct programs need bear no extra overhead - stack recovery is performed only after error detection. Unfortunately this is too late for correction purposes - illegal semantic actions may already have occurred.

---

† We assume initially that S is predicted and then the left $ is immediately popped.

3. Creating Nonnullable Strong LL(1) Grammars

We now show that any Strong LL(1) grammar can be algorithmically transformed into an equivalent nonnullable Strong LL(1) grammar. This of course means Algorithm 2.2 can be considered usable with any Strong LL(1) grammar.

Because we are interested in using Algorithm 2.2 with actual compilers which are driven by Strong LL(1) parsers, we must consider how a Strong LL(1) parser interfaces with the rest of a compiler. Typically this is by use of action symbols [7]. Action symbols represent a fixed (and finite) set of grammar symbols disjoint from the terminal and non-terminal vocabulary. They are allowed to appear anywhere in the right-hand side of a production. In context-free derivations and in parsing they are completely ignored† except that when a production is predicted they are pushed onto the parse stack as part of the right hand side. When an action symbol reaches the top of the parse stack, it is immediately popped and a subroutine (usually termed a semantic routine) is invoked.

Action symbols thus serve to synchronize the parser with the rest of a compiler. Thus given a production $E' \rightarrow + TE'$, we might add an action symbol {Plus} to the right hand side to obtain $E' \rightarrow +T$ {Plus}E'. This would specify that after predicting and recognizing '+ T' we are to invoke a semantic routine (presumably to generate a plus operation) and then resume parsing by recognizing an E'.

In transforming a Strong LL(1) grammar to a nonnullable Strong LL(1)

---

†Thus no explicit provision for action symbols is needed in Algorithms 2.2 and 2.3

grammar we will need to ensure that, for a given input string, the sequence of action symbols encountered using both grammars will be identical. This will guarantee that the same translation will be obtained using either the transformed grammar or the original.

Let K be the set of action symbols used. For $a \in K$ define ACTION_SEQ(a) = a . For $A \in V_n$ define ACTION_SEQ(A) = the sequence of action symbols which would be obtained as A derives λ . For unambiguous grammars (including all Strong LL(1) grammars) this sequence will be unique. Finally for $X_1 \ldots X_m \in (V_n \cup K)^+$ define ACTION_SEQ($X_1 \ldots X_m$)= ACTION_SEQ(X_1)...ACTION_SEQ(X_m) . Except for the provision for action symbols, the following algorithm is essentially that given by Aho & Ullman in [2] .

Algorithm 3.1

Input: G, a Strong LL(1) grammar

Output: G', an equivalent nonnullable Strong LL(1) grammar

[1] Initially set G' = G

[2] FOR EACH production $p = B \rightarrow X_1 \ldots X_m$

SUCH THAT $m \geq 1$ AND $X_1 \ldots X_m \Rightarrow^+ \lambda$ DO

FOR EACH nonterminal A IN p DO

(a) Let all the productions in G with A as a left-hand side be $A \rightarrow \alpha_0 \,|\alpha_i|\ldots|\alpha_n \quad (n \geq 0)$
where $\alpha_0 \Rightarrow^* \lambda$

(b) Let $\overline{A}$ be a new non-terminal added to G'

(c) Add $\overline{A} \rightarrow \alpha_1|\ldots|\alpha_n$ to G'

(d) Write α_0 as $Z_1 \ldots Z_m \quad (m \geq 0)$

(e) Add the following productions to G':
$\{ \overline{A} \rightarrow \text{ACTION\_SEQ}(Z_1 \ldots Z_{i-1})\ \overline{Z}_i\ Z_{i+1} \ldots Z_m \mid$
$\text{First}_1(Z_i) \cap V_t \neq \emptyset$ and $1 \leq i \leq m\}$

(f) Replace all A-productions in G' with

$$A \rightarrow \overline{A} \mid \text{ACTION\_SEQ}(Z_1 \ldots Z_m)\lambda$$

END {FOR EACH}

END {FOR EACH}

[3] Remove all useless non-terminals and productions from G' □

The operation of Algorithm 3.1 is quite straightforward. Whenever a production $A \rightarrow \alpha_0$ is found which can derive λ indirectly (i.e., $\alpha_0 \Rightarrow^+ \lambda$), we create an alternate set of productions which cover derivations through α_0. In particular, A is allowed to derive λ directly or to derive $\overline{A}$. $\overline{A}$ can derive all of $L(\alpha_0)$ less λ. It does this by noting that some non-terminal in α_0 (call it Z_i) must derive the first symbol of some $z \in L(\alpha_0)-\{\lambda\}$. This derivation is provided for by adding a production $\overline{A} \rightarrow \overline{Z}_i\, Z_{i+1} \ldots Z_m$. Z_i can be replaced by $\overline{Z}_i$ because, by construction, $L(\overline{Z}_i) = L(Z_i)-\{\lambda\}$. We can formalize the properties of Algorithm 3.1 in the following.

Theorem 3.2

Assume G' is created from G, a Strong LL(1) grammar, by Algorithm 3.1. Then

(1) $L(G') = L(G)$

(2) G' is nonnullable and Strong LL(1)

(3) The action symbol sequence obtained while parsing any string $x \in L(G)$ via G' will be identical to that obtained while parsing x via G.

Proof

(1), (2). The proof of these two parts parallels exactly the proof of Theorem 8.3 in [2].

(3) This follows from the fact that derivations in G' parallel derivations in G. In fact, they can only differ when a non-terminal A whose productions were changed is expanded. If in G, A was expanded

(3) continued

to an α which can't derive λ, then in G' $A \Rightarrow \overline{A} \Rightarrow \alpha$. Otherwise, α can derive λ. If in the derivation in G it does derive λ we use $A \rightarrow$ ACTION_SEQ(α) λ in G'. By construction, this production yields the same sequence of action symbols as the derivation of λ from A in G does. If $\alpha = Z_1 \ldots Z_m$ derives some non-λ string in G by allowing $Z_1 \ldots Z_{i-1}$ ($i \geq 1$) to derive λ and Z_i to derive some non-λ string then again G' has an analogous derivation sequence. In particular the sequence $A \Rightarrow \overline{A} \Rightarrow$ ACTION_SEQ($Z_1 \ldots Z_{i-1}$) $\overline{Z}_i Z_{i+1} \ldots Z_m$ is used. ACTION_SEQ($Z_1 \ldots Z_{i-1}$) yields the same action symbols as the derivation $Z_1 \ldots Z_{i-1} \Rightarrow^* \lambda$ does in G. We then can iterate the above argument to show that the expansion of $\overline{Z}_i$ to some non-λ string parallels the expansion of Z in G as does the subsequent expansion of $Z_{i+1} \ldots Z_m$ □

It is useful to note that Algorithm 3.1 has the property that only non-terminals and productions involved in the indirect derivation of λ are modified. This means that if a grammar is "very close" to nonnullable form, then it will be perturbed only slightly. This is in sharp contrast to the algorithm presented by Ghezzi [5] which in all cases changes (and expands) the entire grammar.

4. Using Nonnullable Grammars In Practice

Finally, we must consider how closely Strong LL(1) grammars used in practice approximate the nonnullable form required by Algorithm 2.2. Surely a large deviation would be surprising in that indirect derivation of λ seems to be of little use in practice. In fact, examples we have considered confirm this supposition.

Lewis and Rosenkrantz in [6] describe an Algol compiler using a Strong LL(1) parser. The grammar they use has the property† that every

†In [7] this is termed a q-grammar

non-λ production begins with a terminal. Such a grammar is trivially nonnullable.

A Strong LL(1) grammar for a variant of Algol W we considered contained only a few cases of indirect derivation of λ among 180 productions. The most significant of these involved variable qualification in the following grammar fragment:

```
<VAR>    → ID <IDREM>
<IDREM>  →   (<ACTUALS>)
         |   <SUBSCRIPTS><QUALIFIERS>
<SUBSCRIPTS> →   [<EXPR-LIST>]
             | λ
<QUALIFIERS> →  . ID <SUBSCRIPTS><QUALIFIERS>
             | λ
```

The offending production is

<IDREM> → <SUBSCRIPTS><QUALIFIERS> .

Using Algorithm 3.1 (and removing $\overline{A}$-type non-terminals by substituting their sole right-hand sides) this production is replaced with:

```
<IDREM> → [<EXPR-LIST>] <QUALIFIERS>
        | .ID <SUBSCRIPTS><QUALIFIERS>
        | λ
```

The resulting grammar is only marginally larger and is just as readable as the original. Thus in this case it is fair to conclude that transforming this grammar to nonnullable form would not significantly affect its practical value.

5. Conclusions

LL-parsing techniques are very commonly used in practice. Because of size considerations, Strong LL(1) is the LL-technique almost invariably chosen. However if syntactic error-recovery or correction is to be performed effectively, the immediate error detection property must be guaranteed.

We have suggested a simple, compact and efficient algorithm which provides immediate error detection for a broad class of Strong LL(1) grammars. This class of grammars appears to include (with at most minor modifications) those Strong LL(1) grammars used in practice to drive syntax-directed compilers. Further, we have provided an effective algorithm which transforms any Strong LL(1) grammar into the necessary form. Via this transformation, immediate error detection can be provided for any Strong LL(1) grammar.

Several bottom-up parsing methods, including LALR(k) and SLR(k), do not have the immediate error detection property. An investigation of immediate error detection in LALR(k) and SLR(k) parsers is presented in [8].

References

[1] Aho, A. V. and J. D. Ullman, The Theory of Parsing, Translation and Compiling, Vol 1, Prentice-Hall, Sec. 5.1, 334-361, 1972.

[2] Aho. A. V. and J. D. Ullman, The Theory of Parsing, Translation and Compiling, Vol 2, Prentice-Hall, Sec 8.1.2, 674-683, 1973.

[3] Dion, B. A. and C. N. Fischer, An insertion-only error corrector for LR(1), LALR(1), SLR(1) parsers, Technical Report #315, Computer Sciences Department, University of Wisconsin-Madison, 1978.

[4] Fischer, C. N., D. R. Milton and S. B. Quiring, An efficient insertion-only error corrector for LL(1) parsers, Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, 97-103, 1977.
Submitted to Acta Informatica

[5] Ghezzi, C., LL(1) grammars supporting an efficient error handling, Information Processing Letters 3, 174-176, 1975.

[6] Lewis, P. M. II, and D. J. Rosenkrantz, An Algol compiler designed using automata theory, Proc. Symposium on Computers and Automata, Microwave Research Institute Symposia Series, Vol 21, Polytechnic Institute of Brooklyn, N. Y., 75-88, 1971.

[7] Lewis, P. M. II, D. J. Rosenkrantz and R. E. Stearns, Compiler Design Theory, Addison-Wesley, 1976.

[8] Tai, K. C., The recovery of parsing configurations for LR parsers, Department of Computer Science, North Carolina State University, 1978.