A LEAST-COST ERROR CORRECTOR FOR LR(1)-BASED PARSERS

by

Bernard A. Dion

and

Charles N. Fischer

Computer Sciences Technical Report #333

September 1978

COMPUTER SCIENCES DEPARTMENT
University of Wisconsin
1210, West Dayton Street
Madison, Wisconsin 53706

A LEAST-COST ERROR CORRECTOR FOR LR(1)-BASED PARSERS[1]

by

Bernard A. Dion
and
Charles N. Fischer

Computer Sciences Technical Report # 333

September 1978

----

[1]This research was supported in part by the National Science Foundation under grant number MCS 78-02570.

Abstract

An error corrector working with LR(1) parsers and variations such as SLR(1) and LALR(1) is studied. The corrector is able to correct and parse any input string. Upon detection of a syntax error, it operates by deleting Ø or more input symbols and inserting a terminal string that guarentees the first non-deleted symbol to be accepted by the parser. The total correction cost, as defined by a table of deletion and insertion costs, is minimized.

Keywords

Compilers, Parsing, LR(1), Syntactic Errors, Error Correction.

1. INTRODUCTION :

The problem of error recovery and correction in bottom-up context-free parsing has received much attention [2,4,5,6,7,9]. Except for [9], all of these techniques, when faced with certain syntax errors, are forced to skip ahead in the input stream, completly ignoring portions of it. All but [4] and [7] fail to use a cost criterion to guarantee the quality of a correction or recovery operation. Further, in none of the cited work is the issue of time and space complexity dealt with. Indeed in many cases, non-linear behavior is readily obtainable.

Following the work done in [1] and [3], we consider an error corrector which generates "locally least cost" corrections in the presence of *any* syntax errors. This algorithm works with LR(1) parsers and (practical) variations such as SLR(1) and LALR(1) (we term this class *LR(1)-based* parsers). In cases of practical interest, linear time and space complexity can be guaranteed for *all* corrections.

Let G = (V_n,V_t,P,S) be an augmented context-free grammar. All input strings will be terminated by the end marker symbol '\$'. Given an input string $xa_1a_2...a_m\$$ such that $S \Longrightarrow^+ x...$ but $S \not\Longrightarrow^+ xa_1...$, the error corrector will effect a correction by *deleting* symbols $a_1...a_i$ ($\emptyset \leq i \leq m$) and *inserting* a terminal string y such that $S \Longrightarrow^+ xya_{i+1}...$. The values of i and y will be chosen to *minimize* correction costs.

2. RIGHT CONTEXT OF AN ITEM IN A PARSER STATE :

In order to determine a least cost insertion to the immediate left of an error symbol, we need to know which strings can legally appear to the right of an accepted program prefix. We call this set of strings the *right context*.

The theory necessary to construct LR(1)-based parsers is derived from the notion of a viable prefix, i.e. a string which is a prefix of some right sentential form but which does not extend past the handle of that right sentential form. We now introduce the dual notion of viable suffix.

Definiton 2.1 : Let G be an augmented context-free grammar. Assume α is a viable prefix of G. Then β is a viable suffix of G (corresponding to α) iff

(1) $S \Longrightarrow^{*} \alpha\beta$

(2) There does not exist γ such that

$S \Longrightarrow^{*} \alpha\gamma$ and $\gamma \Longrightarrow^{+} \beta$

(i.e. β is fully reduced)

Definition 2.2 : The right context $R(I, s_1 s_2 \ldots s_n)$ of an LR item $I = [A \text{-->} \beta_1 . \beta_2]$ where the contents of the parse stack are $S = s_1 \ldots s_n$ and $I \in s_n$, is the set of strings $\beta_2\gamma \in V^*$ such that $\beta_2\gamma$ is a viable suffix of G corresponding to some viable prefix $\alpha\beta_1$ where

(1) $\alpha\beta_1 = X_1 X_2 \ldots X_{n-1}$, $X_k \in V$ for $k=1,\ldots,n-1$

(2) $GOTO(s_k, X_k) = s_{k+1}$ for $k=1,\ldots,n-1$

We now consider the problem of computing R(I,S). It will be expressed as the concatenation of 3 regular expressions over V. We have $R(I,S) = \beta_2$ cat $l(I,s_n)$ cat $g(I,S)$, where β_2 is the trailing part of I, l is a regular expression denoting the local right context and g is a regular expression denoting the global right context.

Example 2.1 : Consider the following SLR(1) grammar

```
G1: S --> E $
       E --> E + T | T
       T --> a | ( E )
```

Part of G1's CFSM is as follows

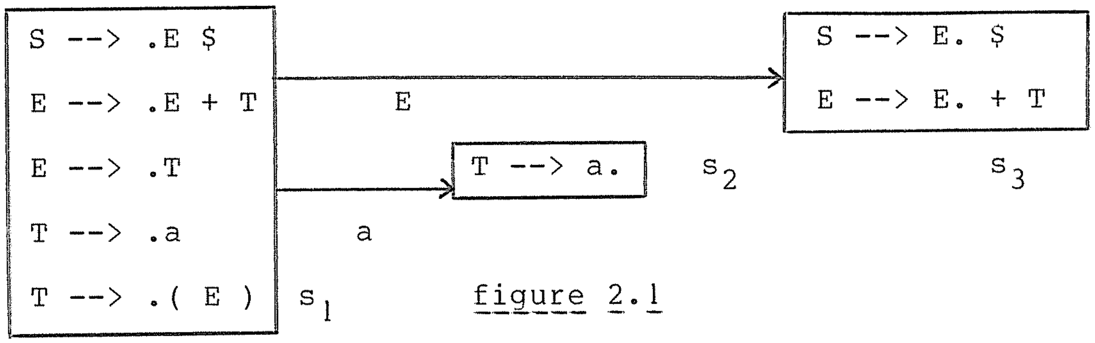

figure 2.1

We now consider state s_1 and draw its closure graph $G(s_1)$ where each item of s_1 is a vertex in $G(s_1)$ and we include a directed edge (I_k, I_1) if $I_k = [B \text{ --> } .\gamma]$ is obtained by prediction from item $I_1 = [A \text{ --> } \alpha.B\beta]$. Edge (I_k, I_1) is labeled β. We thus obtain:

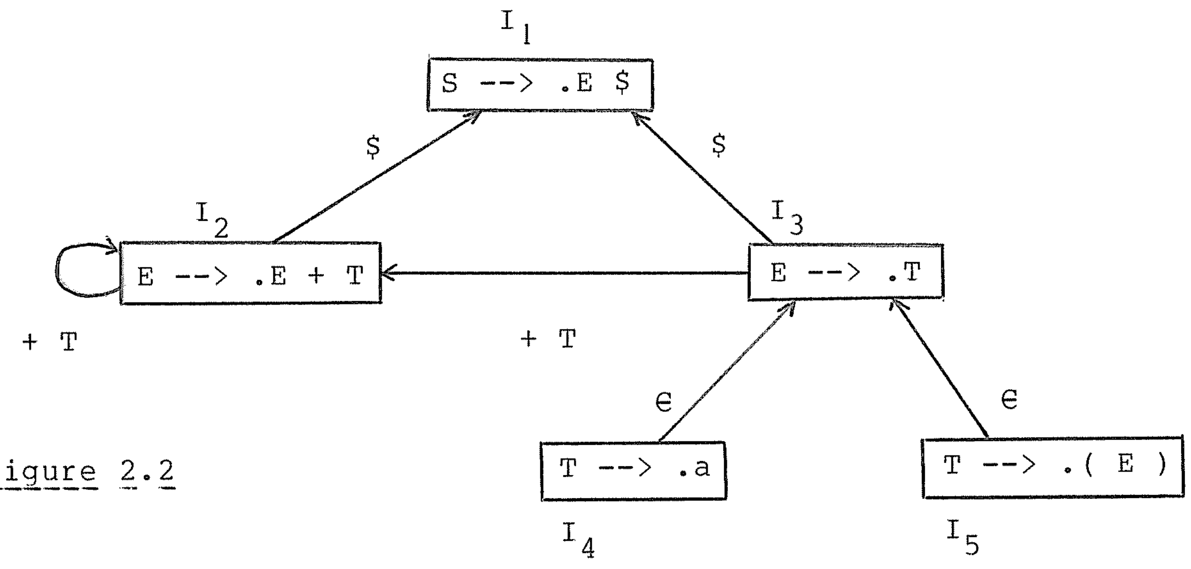

figure 2.2

Now considering any item in a closure graph, its local right context is obtained by concatenating labels on a path from this item to any basis item. For example $l(I_4, s_1) = \{+T\}^* \text{ cat } \{\$\}$ which corresponds to paths I_4, I_3, I_1 and $I_4, I_3, I_2, \ldots, I_2, I_1$. We denote by $l(I_i, I_j, s)$ the regular expression of all paths between items I_i and I_j in $G(s)$. For example $l(I_5, I_3, s_1) = \{\epsilon\}$. We also denote by $L(l(I,s))$ and $L(l(I,J,s))$ the set of terminal strings derivable from $l(I,s)$ and $l(I,J,s)$.

The part of the right context which cannot be obtained locally (i.e. looking at the closure graph of the top stack state) is termed global right context. It is a function of the parse stack. Still considering G_1, assume the parse stack is s_1s_2. We have $g([T \dashrightarrow a.], s_1s_2) = \{+T\}^* \underline{cat} \{\$\}$. This is obtained by considering the local right context of the predecessor of [T --> a.] in s_1. In general, the global right context can be obtained by concatenating appropriate local right contexts for each state on the parse stack.

3. AN INSERTION-ONLY ERROR CORRECTOR :

As developed in [1], we first present a restricted version of our algorithm where we only consider the insertion of a terminal string as a potential correction. More precisely, we assume a given insertion cost vector C where C(a) is given for all a $\in V_t$. The cost of inserting a terminal string is the sum of the insertion costs of the terminal symbols in the string. We now formulate the problem in the following way: given an input string xa... such that $S \Longrightarrow^+ x\ldots$ and $S \not\Longrightarrow^+ xa\ldots$ find an optimal solution $y \in V_t^+$ to

$$\min_{y' \in V_t^+} \{ C(y') \mid S \Longrightarrow^+ xy'a\ldots \} \qquad (3.1)$$

Of course the existence of such a string y is not guaranteed in general. Languages for which y always exists are termed insert-correctable [2]. Interestingly enough, it can be shown that a programming language such as ALGOL 60 is insert-correctable (modulo a very minor modification). In [1] an algorithm is presented that tests if an LR(1) grammar is insert-correctable.

3.1 Immediate Error Detection Property:

A parsing algorithm is said to have the *immediate error detection property* (IEDP) iff an error is detected as soon as an erroneous input symbol is first encountered. As we will see later, our error corrector requires the IEDP.

It is a well-known fact that canonical LR(1) parsing has the IEDP. However standard SLR(1) and LALR(1) parsers don't. Thus we have to modify these algorithms slightly. Because of the fact that they use follow sets which are approximations to exact lookaheads, an erroneous symbol may cause wrong reductions. In order to guarantee the IEDP, we save parser moves in a queue until the next input symbol is accepted by the parser. If it turns out that the input symbol is erroneous, the parse stack is restored to the condition it had when this symbol was first seen, using information stored in the queue.

In the worst case this buffering technique can require $O(n^2)$ extra time to process an input of size n. However linearity is achieved in the case of a *bounded depth parse stack*, which is almost always used in practice.

3.2 Error Correction Algorithm :

We are now ready to discuss the LR_INSERT function (algorithm 3.1, p.8) which is called upon detection of a syntax error (the parse stack being restored, if necessary). The input to the function is 'a', the error symbol and $S = s_1 s_2 \ldots s_p$, the parse stack. Its output is LR_INSERT which is a solution to problem 3.1 .

LR_INSERT processes the parse stack in a top-down fashion, creating a *stage* for each stack state. A stage is a collection of 'LC' values, one per basis item in the corresponding state. If item I_i is in the basis of some state then LC_i is a least cost terminal string whose insertion allows us to complete the recognition of I_i when the parser is restarted in state s_p (LC_i is termed a <u>least-cost completor</u> of I_i). As we process the stack we keep track of 2 stages only : CURSTAGE corresponding to state s_k and PREDSTAGE corresponding to state s_{k-1}.

Let us introduce some notation that is needed :

(1) $S(\alpha)$ is a least cost string derivable from α.

(2) Insert(β_i,a) is a solution to the following problem

$$\min_{t' \in V_t^*} \{ C(t') \mid \beta_i \overset{*}{==>} t'a\ldots \}$$

If no solution exists, Insert yields '?', a special symbol with infinite insertion cost.

The function starts by initializing INSERTION to '?'. It initializes LC values for the top stack state and checks if a correction can be obtained from the trailing part of a basis item in this state (lines 2-6). It then processes the parse stack until no lower cost insertion can be found (lines 8-34). At each iteration of the while loop (line 8), LC values of PREDSTAGE (corresponding to state s_{k-1}) are deduced from LC values of CURSTAGE (corresponding to state s_k). This is done by linking basis items in these two states : a basis item in s_k is linked to its predecessor in s_{k-1} (line 12) which, in turn, is linked to basis items in s_{k-1} following paths in the closure graph of s_{k-1} (line 14). It also has to check for the possibility of discovering a lower cost insertion from the local right context of a closure item in s_{k-1} and it updates INSERTION if this is the case (lines 22-26).

To allow an efficient implementation of Algorithm 3.1, many of the values needed by the algorithm can be computed in advance and tabled. This is detailed in [1].

Example 3.1 : Using grammar G1, assume that the input string is 'aa\$' and unit insertion costs are in effect. When an error is detected we are in the following configuration : stack = $s_1 s_3$, error symbol = 'a'. Considering s_3, we obtain LC_1 = S(\$) = '\$' and LC_2 = S(+T) = '+a' (line 3). Further, INSERTION = Insert(+T,a) = '+' (line 5). Since C(+) is less than both C(LC_1) and C(LC_2) (by convention, C(\$) is infinite) the computation immediatly terminates (test of line 8) with a correction of 'aa\$' into 'a+a\$'. The error corrector thus attempts to effect corrections using local context only.

When necessary, however, Algorithm 3.1 considers just enough global right context to guarantee that the lowest cost correction possible is effected. Thus given an input string of 'a++a\$', error correction would commence in a state whose sole basis item is [E --> E+.T]. Now LC_1 = S(T) = a and INSERTION = Insert(T,+) = '(a'. Since C('(a') > C('a'), we would examine states below the uppermost and discover 'a' to be the least-cost INSERTION value.

```
Function LR_INSERT(a, S) : terminal_string ;
begin { initialize error correction using top state }
    k := p; INSERTION := ? ; CURSTAGE := STAGE($s_p$) ;
    for all i such that [$A_i$ --> $\alpha_i . \beta_i$] $\in$ Basis($s_p$) do
       CURSTAGE.$LC_i$ := S($\beta_i$) ;
       if C(Insert($\beta_i$,a)) < C(INSERTION)
           then INSERTION := Insert($\beta_i$,a) fi
    od ;
    { now process stack until no lower cost insertion possible }
    while $\exists$ i $\ni$ C(CURSTAGE.$LC_i$) < C(INSERTION) and k > 1 do
       PREDSTAGE := STAGE($s_{k-1}$) ; { LC values are initially = '?'}
       for all $I_l$ $\in$ Basis($s_k$) such that
               C(CURSTAGE.$LC_l$) < C(INSERTION) do
          let $I_m$ be the predecessor of $I_l$ in $s_{k-1}$ ;
          if $I_m$ $\in$ Closure($s_{k-1}$) then
             for all $I_j$ $\in$ Basis($s_{k-1}$) such that l($I_m$,$I_j$,$s_{k-1}$) $\neq$ $\emptyset$
             do
                let y be a least cost string in L(l($I_m$,$I_j$,$s_{k-1}$)) ;
                if C(CURSTAGE.$LC_l$) + C(y)
                   < C(PREDSTAGE.$LC_j$) then
                      PREDSTAGE.$LC_j$ := CURSTAGE.$LC_l$ cat y
                fi
             od ;
             let t be a solution to
                 min  { C(t') | t'a... $\in$ L(l($I_m$,$s_{k-1}$)) };
                 t' $\in$ $V_t^*$
             if C(CURSTAGE.$LC_l$) + C(t) < C(INSERTION)
                   then INSERTION := CURSTAGE.$LC_l$ cat t fi
          else { $I_m$ $\in$ Basis($s_{k-1}$) }
             if C(CURSTAGE.$LC_l$) < C(PREDSTAGE.$LC_m$)
                 then PREDSTAGE.$LC_m$ := CURSTAGE.$LC_l$
             fi
          fi
       od ;
       CURSTAGE := PREDSTAGE ; k := k-1
    end while ;
    LR_INSERT := INSERTION
end LR_INSERT.
```

algorithm 3.1

4. EXTENDED ERROR CORRECTION :

We now extend the model to include deletions as specified in section 1. Let $D(a)$ for $a \in V_t$ be the cost of deleting a. Assume $D(\$) = +\inf$. Again assuming that $S \Longrightarrow^{+} x\ldots$ but $S \not\Longrightarrow^{+} x a_1 a_2 \ldots a_m \$$, we want to find a solution to the following problem :

$$\min_{0 \le i \le m} \{ \min_{y' \in V_t^*} \{ C(y') + D(a_1 \ldots a_i) \mid S \Longrightarrow^{+} x y' a_{i+1} \ldots a_m \$ \} \} \quad (4.1)$$

This can be implemented very simply by calling Algorithm 3.1 repeatedly with error symbols $a_1, a_2 \ldots$ until we reach a situation where the cumulative deletion cost $D(a_1 \ldots a_{i+1})$ is larger than $C(y') + D(a_1 \ldots a_i)$ as obtained in the previous step. However this procedure has an $O(|x|^2)$ worst case runnning time because it may have to reprocess each input symbol $O(|x|)$ times.

Now assume that we do the following preprocessing of $a_1 \ldots a_m$: we maintain a vector FIRST_INPUT : $V_t \rightarrow \{1,\ldots,m\} \cup \{\text{ABSENT}\}$ which points to the first occurence, if any, of a terminal in $a_1 \ldots a_m$. Also, each terminal in $a_1 \ldots a_m$ is labeled with DC such that $DC(i) = D(a_1 \ldots a_{i-1})$ and $DC(\text{ABSENT}) = +\inf$. We now find $c \in V_t$ which is a solution to

$$\min_{c \in V_t} \{ DC(\text{FIRST\_INPUT}(c)) + C(\text{LR\_INSERT}(c,S)) \} \quad (4.2)$$

where S is the parse stack and LR_INSERT is the terminal string that is computed by algorithm 3.1.

The basis for the correctness of this procedure is the fact that we only need to consider deletions up to the first occurence of any terminal in order to find the least cost correction. This calculation can be done in constant time per error if a bounded depth parse stack is used [1].

In practice, the preprocessing assumed by (4.2) would probably be done incrementally. That is, we would first compute the cost of corrections involving 0 deletions, then 1 deletion, etc, calling LR_INSERT at most once for a given terminal. As soon as the best known correction is no more expensive than the cumulative deletion cost, processing can be terminated.

Returning to example 3.1, assume all insertions and deletions are of unit cost. While correcting 'aa$', corrections involving 0 deletions are first considered. The best of these is insertion of '+' at a cost of 1. Since deleting 'a' would also cost 1, no further processing occurs. If however, all insertions are modified to cost 2 each, after the first step the best known correction has a cost of 2. Now deletion of 'a' is explored. This deletion costs only 1 and as no subsequent insertions are needed, a lower cost correction is discovered. Thus a correction of 'aa$' into 'a$' rather than 'a+a$' is obtained.

Normally costs are set so that deletions are more expensive than insertions (to encourage corrections which "build upon" existing input strings). Under such circumstances, the above technique of considering deletions incrementally seems a particularly good way of finding least cost corrections quickly and efficiently.

We finally summarize the properties of this algorithm with the following theorems.

<u>Theorem 4.1</u> : Assume that for some cfg G, $x\ldots \in L(G)$ but $xa\ldots \notin L(G)$. Further assume that while attempting to parse $xa\ldots$ an LR(1)-based parser invokes the extended error corrector as soon as 'a' is encountered. Then it will delete i input symbols and then insert a terminal string y' where i and y' satisfy problem

4.1.

Theorem 4.2 : Assume an LR(1)-based parser using the procedure outlined in 4.2 as an error corrector processes x$. Then it reqires

(a) at most $O(|x|^2)$ time and $O(|x|)$ space in the general case.

(b) at most $O(|x|)$ time and space if a bounded depth parse stack is assumed.

(c) at most $O(|x|)$ time and space if a canonical LR(1) parser is assumed.

The $O(|x|^2)$ time bound arises in the general case because for each of $O(|x|)$ possible errors, it may take $O(|x|)$ time to restore the parse stack (as per section 3.1) and then execute Algorithm 3.1. For the expected case of a bounded depth parse stack, we can process each error in constant time. Alternately, if we use a canonical LR(1) parser no stack restoration is needed. A more complex version of Algorithm 3.1 (see [1]) can then be used to guarantee linearity. Thus in cases of both theoretical and practical interest linear LR(1)-based correctors are available.

5. CONCLUSIONS AND SIGNIFICANCE :

The LR-corrector presented above has both theoretical and practical significance. Theoretically, the algorithm can be shown to operate correctly on any input string. A least-cost correction is guaranteed and, in cases of special interest (e.g., bounded depth parse stack), linearity can be established.

On the practical side, preliminary experience indicates that our LR-corrector can be used satisfactorily with most LR-driven compilers. As developed in [1], values needed to drive the corrector can be precomputed and stored in secondary storage. This allows the algorithm to operate quite efficiently with a rather small primary storage requirement.

This research can be extended in several ways. For example, it can be noted that our definition of least cost correction is a very _local_ one since it is concerned with finding an insertion which allows the first non-deleted input symbol to be accepted by the parser. We believe it is worhtwhile to develop more _global_ error correction techniques. Recent work by Penello and DeRemer [6] shows how a forward move can be used by an LR-based parser to condense information on that part of the input string which is to the right of the error symbol, thus providing potentially unbounded lookahead. We plan to investigate the possibility of using such information in the context of least-cost correction.

In [8] Watt shows how attribute grammars can be parsed and evaluated using LR techniques. The possibility of using attribute information (such as types, symbol tables, etc.) to aid in choosing corrections is a very interesting topic of future research.

References

[1] Dion B.A. and C.N. Fischer
An Insertion-only Error Corrector for LR(1),LALR(1),SLR(1) Parsers.
Technical Report #315. February 1978. University of Wisconsin.

[2] Druseikis, F. C. and G.D. Ripley
Extended SLR(k) Parsers for Error Recovery and Repair.
Technical Report. University of Arizona, 1977.

[3] Fischer, C.N. , D.R. Milton and S.B. Quiring
An Efficient Insertion-only Error Corrector for LL(1) Parsers.
Conference Record of the 4th ACM Symposium on Principles of Programming Languages, pp. 97-103.
Submitted to Acta Informatica.

[4] Graham, S.L. and S.P. Rhodes
Practical Syntax Error Recovery.
Communications of ACM. November 1975. Vol.18 no.11 , pp 639-650.

[5] James L.R.
A Syntax Directed Error Recovery Method.
Technical Report CSRG-13. May 1972. University of Toronto.

[6] Penello T.J. and F. DeRemer
A Forward Move for LR Error Recovery.
Conference Record of the 5th ACM Symposium on Principles of Programming Languages. January 1978.

[7] Tai, K.C.
Syntactic Error Correction in Programming Languages.
Ph.D. Thesis. Department of Computer Sciences. Cornell Univ., 1977.

[8] Watt, D.A.
The Parsing Problem for Affix Grammars.
Acta Informatica 8,1-20 (1977) pp. 1-20.

[9] Mickunas, M.D. and J.A. Modry
Automatic Error Recovery for LR Parsers.
Communications of ACM. Vol.21, no.6, June 1978, pp.459-465.