A LOCALLY LEAST-COST LR-ERROR CORRECTOR

by

C. N. Fischer
B. A. Dion
J. Mauney

Computer Sciences Department Report #363

August 1979

A LOCALLY LEAST-COST LR-ERROR CORRECTOR

by

C. N. Fischer
B. A. Dion
J. Mauney

Computer Sciences Department Report #363

August 1979

A Locally Least-Cost LR-Error Corrector[1]

C. N. Fischer

B. A. Dion[2]

J. Mauney

University of Wisconsin - Madison
Madison, Wi. 53706

[1] Research supported in part by National Science Foundation Grant MCS78-02570

[2] Present address: CII-Honeywell Bull, 75960 Paris Cedex 20, France

Abstract

An error corrector usable with LR(1) parsers and variations such as SLR(1) and LALR(1) is studied. The algorithm is able to correct and parse any input string. It chooses locally least-cost repair operations (as defined by the user) in correcting all syntax errors. Moreover, the error corrector can be generated automatically from the grammar and a table of terminal correction costs. Correctness, local optimality and linearity of the algorithm are established. Implementation and test results are presented.

Keywords and Phrases:

Error correction, error recovery, LR parsing, diagnostic compilers, syntax errors

CR Categories: 4.12, 4.42, 5.23

1. Introduction

The problem of correcting and recovering from syntax errors in context-free parsing has received much attention ([1], [5], [6], [8]-[36]). Known solutions, however, usually contain some rather serious drawbacks. Some ([4], [6], [18], [19], [27]) are essentially ad-hoc, requiring the use of hand-coded recovery routines. Others ([9], [16], [17], [18], [19], [22], [26], [27], [30], [35]) when faced with certain syntax errors are forced to skip ahead, completely ignoring portions of the input. Some methods ([1], [19], [28], [29] must be considered impractical in that they have non-linear space or time bounds. Yet others ([19]-[22], [27], [31]) when given a choice of possible repairs make an arbitrary (and all too often unreasonable) choice.

We consider an error-correction algorithm which works with LR(1) parsers and such popular variations as SLR(1) and LALR(1). We shall term such parsers LR(1)-based. The corrector has many desirable properties. It requires at most linear time and space. It produces a syntactically correct program from *any* input string. It can be automatically generated from a cfg and a supplied vector of terminal insertion and deletion costs. It always chooses a locally optimal correction (as defined by the cost vectors). This allows the choice of corrections to be "fine-tuned" by merely adjusting the costs. Because the choice of corrections is determined solely by the *language* being parsed and the correction costs (and not by the underlying grammar or details of the correction algorithm), a very high level model of the correction process is obtained. This is especially important as it allows a user to predict (and, via costs, alter) the response a given error will elicit without knowing anything of the actual correction mechanism.

Even if a compiler is not designed to perform syntactic error-correction, it must be able to recover from syntax errors. Because of its simplicity, efficiency, and robustness, this error-corrector is well-suited for use as an error-recovery technique. When parsing tables are generated, error-correction tables (assuming unit correction costs) can also be automatically provided. When a syntax error is detected, the error-correction algorithm can be invoked *invisibly* to "reset" the parser to a state in which syntax analysis may be restarted. We are therefore able to provide an error-recovery capability as an *integral part* of an LR(1)-based parser.

In what follows, it is assumed that the reader is familiar with the basic notions of grammars and parsing [2]. The empty (or null) string is denoted by λ. *cat* denotes string catenation.

2. Notation and Preliminaries

In general, our corrector may need to perform corrections at any point including the very end of an input string. It is therefore necessary to use an augmented cfg. Let $G = (V_n, V_t, P, Z)$. Then the augmented cfg $G' = (V_n \cup \{Z'\}, V_t \cup \{\$\}, P \cup \{Z' \rightarrow Z\$\}, Z')$, where $\$ \notin V_t$, $Z' \notin V_n$. All input strings will be terminated by the endmarker symbol, \$. We shall consider all cfgs to be augmented and denote $V_t \cup \{\$\}$ by $\hat{V}_t$, $V_n \cup \{Z'\}$ by $\hat{V}_n$. Similarly, $V = V_n \cup V_t$ and $\hat{V} = \hat{V}_n \cup \hat{V}_t$.

Given an input string $xb_1b_2 \ldots b_n$ ($x \in V_t^*$, $n \geq 1$, $b_1, b_2, \ldots b_n \in \hat{V}_t$) such that $Z' \Rightarrow^+ x\ldots$ but $Z' \not\Rightarrow^+ xb_1\ldots$, our correction algorithm will delete the next i input symbols ($i \geq 0$) and then will insert a string $y \in V_t^*$ such that $Z' \Rightarrow^+ xyb_{i+1}\ldots$. Further, i and y will be chosen so as to minimize the associated correction cost (i.e., the correction will be <u>locally optimal</u>). Note that x, the input prefix already accepted, is <u>never changed</u>. This allows symbols in x to be translated as they are accepted without fear that subsequent errors may force translation steps to be "undone." Indeed, in languages like Pascal which are specifically designed to allow programs to be compiled in <u>one pass</u>, this assumption is especially important as it allows error correction to be performed as translation proceeds (which is absolutely necessary in a one pass compiler).

Note too that our corrector is intentionally <u>local</u> in scope. That is, upon detection of an error it examines just enough of the remaining input to determine a least-cost correction which will allow the parser to accept the next (non-deleted) input symbol. This allows error correction to be reasonably simple and efficient. Other correction techniques ([16], [17], [31], [33]) advocate a <u>forward move</u> strategy in which significant amounts of the remaining input are examined to determine the "most appropriate" correction. Because more context is examined, such techniques can sometimes yield better corrections. However such embellishments can be rather involved and can negatively affect other phases of the compilation process[3]. Further, as described in section 6, tests indicate that our local approach yields satisfactory repairs in almost all error situations. Thus the correction model we adopt seems to be a good compromise between the generality needed to guarantee reasonable corrections and the simplicity needed to ensure efficient and economic implementations.

In order to have our corrector operate properly, a parser must detect an error upon first encountering an erroneous input symbol. That is, the parser must have the <u>Immediate Error Detection</u> (IED) Property. It is well known that canonical LR(1) parsers have the IED property [2]. However because such popular

---

[3] E.g., one-pass compilation can be precluded [31].

LR(1)-based techniques as SLR(1) and LALR(1) use approximations to exact lookaheads [7], they can perform erroneous reductions when an illegal symbol appears as the lookahead. Thus they do not possess the IED property.

As an example of the problems which can arise, consider the following SLR(1) cfg:

```
G1:   Z' --> E $
      E  --> T E'
      E' --> + T E'  |  λ
      T  --> a  |  ( E )
```

Now consider an SLR(1) parser for G1 processing 'a)...$'. The following parsing sequence will occur:

Step	Lookahead	Parser Action	Accepted Program Prefix
1	a	Shift	a
2	)	Reduce [T --> a]	T
3	)	Reduce [E' --> λ]	TE'
4	)	Reduce [E --> TE']	E
5	)	Error	E

At this point only '$' can be read and the <u>only possible</u> correction is to delete <u>all</u> remaining input up to the '$'.

To handle this problem, we propose the addition of a <u>reduction stack</u> to SLR(1) and LALR(1) parsers. This stack would hold all the reduction moves induced by a given lookahead. If a lookahead is shifted (and thus verified as correct), the reduction stack is simply cleared. If it is found that the lookahead is erroneous, the reductions saved can be popped and

used to restore the parse stack to that configuration extant when the lookahead was first used. Thus in the above example the three erroneous reductions saved on the reduction stack could be used to restore the parser configuration of step 1. In this configuration a wide variety of repair operations are possible, including deletion of ')' or (if it is cheaper) insertion of '*(a'. In what follows, we shall assume that all LR-based parsers used with our correction algorithm guarantee the IED property.

3. A Least-Cost Insertion Algorithm

In this section, we define a locally least-cost insertion algorithm for LR(1)-based parsers. That is, given an error sitation in which $Z' \Longrightarrow^{+} x\ldots$ and $Z' \not\Longrightarrow^{+} xa\ldots$ ($x \in V_t^*$, $a \in \hat{V}_t$), the algorithm will find a least-cost $y \in V_t^+$ such that $Z' \Longrightarrow^{+} xya\ldots$. For a class of cfl's termed <u>insert-correctable</u> ([12], [14]), such a strategy can correct <u>any</u> syntax error. However in general it may occur that no such y exists (i.e., 'a', the error symbol, must be deleted). In this case, the algorithm will return '?' as an indication of failure.

The correction algorithm will require two auxiliary functions, S and E. These functions rely on an insertion-cost function C: $C(\lambda)$ is defined to be $\emptyset$; for $a \in \hat{V}_t$, $C(a) \geq \emptyset$ is supplied as an a priori value[4], and for $w = X_1 \ldots X_m \in \hat{V}_t^*$, $C(w) = C(X_1)+\ldots+C(X_m)$. The value of a string-valued function whose result is undefined is denoted as "?" where $? \notin \hat{V}$. $C(?)$ is defined to be ∞. We can use the C function to define the Min function over terminal strings:

$$\text{For } x,y \in \hat{V}_t^* \cup \{?\}$$
$$\text{Min}(x,y) = \text{If } C(y) < C(x) \text{ Then } y \text{ Else } x.$$

For $X \in \hat{V}$, define $S(X)$ to be an optimal solution to:

$$\text{Min} \quad \{y \in \hat{V}_t^* \mid X \Longrightarrow^* y\}$$

In other words, $S(X)$ identifies the least-cost terminal string derivable from X. Further, $S(X_1 \ldots X_m) = S(X_1)$ cat ... cat $S(X_m)$ ($m \geq \emptyset$, $X_i \in \hat{V}$). The insertion-cost function C can now be extended to strings: $C(\gamma) = C(S(\gamma))$.

For $\gamma \in \hat{V}^*$ and $a \in \hat{V}_t$, we define $E(\gamma,a)$ to be an optimal solution to:

$$\text{Min} \quad \{y \in V_t^* \mid \gamma \Longrightarrow^* ya\ldots\}$$
$$\text{If } \gamma \not\Longrightarrow^* \ldots a \ldots, \text{ then } E(\gamma,a) = ?$$

[4]Since \$ is assumed to be guaranteed as the last input symbol, it will never be inserted during correction. Thus C(\$) is not strictly needed, but is included to simplify notation.

$E(\gamma,a)$ determines a least-cost prefix which allows 'a' to be derived from γ. Note that for $\gamma = X_1 \ldots X_m$ ($m>1$, $X_i \in \hat{V}$), $E(\gamma,a)$ is equal to[5]:

$$\underset{\emptyset \leq i < m}{\text{Min}} \; S(X_1 \ldots X_i) \;\underline{\text{cat}}\; E(X_{i+1},a)$$

The latter formulation is useful when computing actual E values. Algorithms which compute the S and E functions are detailed in [12].

The S and E functions were first defined for use with the LL(1) corrector described in [12]. In fact, these functions led to an almost trivial least-cost insertion algorithm for LL(1) parsers: Given an LL(1) parse stack of $X_n \ldots X_1$ and an error symbol of 'a', a least-cost insertion is simply $E(X_n \ldots X_1,a)$. Such simplicity is possible because the LL(1) parse stack explicitly stores those grammar symbols which the remaining input is expected to match. In LR-based parsers such information is also available (otherwise incorrect input could not be recognized), but it is in a considerably less useful form. It is therefore necessary to review a few concepts of LR parsing.

A state of the Characteristic Finite State Machine (CFSM) [7] used by LR parsers corresponds to a (finite) set of LR

[5]Assume x cat ? = ? for all $x \in \hat{V}_t^*$.

items[6] of the form $[A \dashrightarrow \alpha\cdot\beta]$ where $A \dashrightarrow \alpha\beta$ is a production. Item $[A \dashrightarrow \alpha\cdot\beta]$ represents the fact that the prefix α has already been recognized and if β matches a prefix of the remaining input, then an application of production $A \dashrightarrow \alpha\beta$ may have been discovered. CFSM states represent sets of items corresponding to all the productions which might match the current input symbols.

The items in a CFSM state s can be partitioned into two disjoint sets [7]:

$s = \text{Basis}(s) \cup \text{Closure}(s)$

For $s_\emptyset$ (the initial or start state)
$\text{Basis}(s_\emptyset) = \{[Z' \dashrightarrow \cdot Z\$]\}$

For $s \neq s_\emptyset$
$\text{Basis}(s) = \{I = [A \dashrightarrow \alpha\cdot\beta] \mid I \in s, \alpha \neq \lambda\}$

For all CFSM states s
$\text{Closure}(s) = \{I = [B \dashrightarrow \cdot\gamma] \text{ or } I = [C \dashrightarrow \lambda\cdot] \mid I \in s\}$

Basis items are created from items in a previous state by a shift operation. That is, if s_i, the current parse stack top, contains $[A \dashrightarrow \alpha\cdot X\beta]$ and $X \in \hat{V}$ is shifted, then $[A \dashrightarrow \alpha X\cdot\beta] \in \text{Basis}(s_{i+1})$ where s_{i+1} is the new stack top after the shift. Closure items are created (directly or indirectly) by prediction operations. That is, if $[B \dashrightarrow \alpha\cdot C\delta] \in s$ for $C \in \hat{V}_n$, then $([C \dashrightarrow \cdot\gamma]$ or $[C \dashrightarrow \lambda\cdot]) \in \text{Closure}(s)$.

---

[6]The lookahead component found in LR(1) items is not needed for correction purposes and can be ignored.

We can now consider the problem of finding a least-cost insertion which corrects a syntax error. Assume the LR parse stack is $s_1 \ldots s_i$ and the error symbol is 'a'. We wish to compute a least-cost string "Insert" such that Insert cat 'a' will be accepted by the parser.

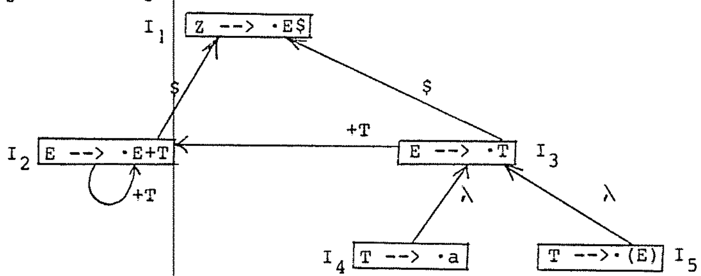
First, we examine items in $\text{Basis}(s_i)$ where s_i is the stack top. If $[A \dashrightarrow \alpha\cdot\beta] \in \text{Basis}(s_i)$, a possible value for Insert is $E(\beta,a)$. It may however happen that $E(\beta,a)$ = '?' or that a lower cost correction can be obtained by examining a predecessor of $[A \dashrightarrow \alpha\cdot\beta]$. To do this, we first need to find a least-cost string which will allow $[A \dashrightarrow \alpha\cdot\beta]$ to be fully matched (so that a predecessor of it can match the error symbol). This is just $S(\beta)$ and is termed a least-cost completor (LCC) of the item. Note that any correction in which this item participates will begin (i.e., be prefixed by) this LCC value.

To consider predecessors of item $[A \dashrightarrow \alpha\cdot\beta]$, we examine state s_j where $j = i - |\alpha|$. There $[A \dashrightarrow \cdot\alpha\beta] \in \text{Closure}(s_j)$. Now in s_j, $[A \dashrightarrow \cdot\alpha\beta]$ may have been predicted by more than one item and therefore may have many different predecessors, all of which need to be considered. To allow all predecessors of a given item to be considered, we construct, for each state s, a closure graph, CG(s). Nodes of CG(s) are the items of s. If item $K = [B \dashrightarrow \cdot\gamma]$ can be obtained from item $J = [A \dashrightarrow \alpha\cdot B\beta]$, then an arc from K to J is created in CG(s) and is labelled with

β. β represents those symbols which can be matched once item K is fully recognized (i.e., completed). As an example, consider tne following cfg:

```
G2:  Z --> E $
     E --> E + T | T
     T --> a | ( E )
```

$CG(s_\emptyset)$, where $s_\emptyset$ is the start state, is:



Because, e.g., there is a path from item I_5 to item I_1, we know that I_1 is a predecessor of I_5. Further, because 'λ' cat '\$' = '\$' labels a path from I_5, we know that once I_5 is completed, a '\$' can be read.

Let $LP(s_i, I, J)$ be the set of all labelled paths from item I to item J in $CG(s_i)$. In general this will be a regular set ([2] Sect. 2.2). Similarly, let $LP(s_i, I)$ be the set of all labelled paths from I to any node in $CG(s_i)$. This too is a regular set. For example, $LP(s_\emptyset, I_5, I_1) = \{+T\}^*\{\$\}$ and $LP(s_\emptyset, I_5) = \{+T\}^*(\{\$\} \cup \{\lambda\})$. We extend the S and E functions to labelled paths in the obvious manner:

Let RS be any regular set over $\hat{V}$.
Then[7]

(1) S(RS) is an optimal solution to
$\quad$ Min $\{y \in \hat{V}_t^* \mid \gamma \in RS \text{ and } y = S(\gamma)\}$

(2) E(RS,a) is an optimal solution to
$\quad$ Min $\{y \in \hat{V}_t^* \cup \{?\} \mid \gamma \in RS \text{ and } y = E(\gamma,a)\}$

Returning to our error correction scenario, recall that we are interested in considering predecessors of item I = [A --> $\cdot\alpha\beta$] in state s_j. It may be that the error symbol 'a' can be derived from such a predecessor. A least-cost way of doing this is to insert LCC cat $E(LP(s_j, I), a)$. That is, LCC completes item I and the E value generates 'a' from I's predecessors in s_j. This value is assigned to Insert if it is better (i.e., cheaper) than the current value.

Alternately, it may be cheaper to generate 'a' from a predecessor of some basis item K of s_j (i.e., from a still deeper parse stack state). To do this, we first must complete item K. LCC completed item I and $LP(s_j, I, K)$ contains those symbols expected once I is completed. Thus LCC cat $S(LP(s_j, I, K))$ can be used to complete item K. Then we can proceed to the state in which K was first predicted and repeat the above process.

Our insertion algorithm will then operate as follows. First, basis items in the top stack state are considered and a

---

[7]Assume $S(\phi)$ = ? and $E(\phi,a)$ = ?.

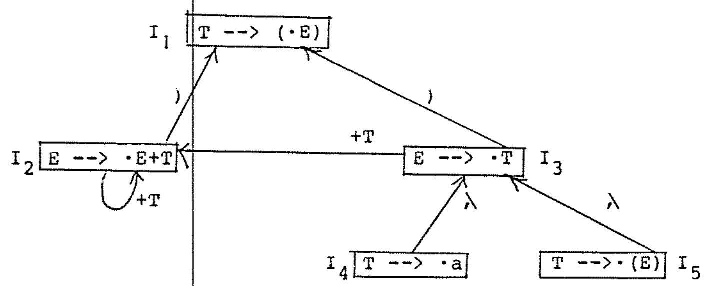
first approximation to Insert is determined. If any basis items have an LCC value cheaper than this value of Insert, their predecessors are considered. This continues until all LCC values are as expensive as Insert or until the entire parse stack is processed.

We can now formally present the insertion algorithm. Let Pred(s,I) return that item J in state s which is a predecessor[8] of item I. For example, Pred(s, [A --> αX·β]) = [A --> α·Xβ]. Further, let LCC(i) be an array of strings corresponding to the LCC values of basis items for that CFSM state at position i in the parse stack. Thus LCC(i,I) gives the LCC value of item I in state s_i.

[8]Pred(s,I) will be undefined for some values of s and I.


```
function LR_Insert(s1...sp,a) : Insert;
 { s1...sp is the LR parse stack (sp at the top),
   a is the error symbol,
   Insert is the least-cost insertion to be computed }
 { Initialization }
 k := p; { k is a stack pointer }
 Insert := ?; { No legal insertion is known yet }
 { Start with the stack top }
 for all items I = [A_I --> α_I·β_I] ∈ Basis(sp) do
   LCC(p,I) := S(β_I);
   Insert := Min(Insert,E(β_I,a));
 end {For all}
 { Now process the rest of the stack until no
   cheaper value of Insert is possible }
 while there exists an item I such that
  C(LCC(k,I)) < C(Insert) and k>1 do
   LCC(k-1) := ? { Initialize next deeper state }
   for all items I ∈ Basis(s_k) such that
    C(LCC(k,I)) < C(Insert) do {Process predecessors }
     J := Pred(s_k-1,I);
     if J ∈ Closure(s_k-1)
     then {Try to update LCC values in s_k-1 }
       for all K ∈ Basis(S_k-1) do
         LCC(k-1,K) :=
          Min(LCC(k-1,K), LCC(k,I) cat S(LP(s_k-1,J,K)));
       end {For all}
       { Now try to find a lower cost insertion }
       Insert := Min(Insert, LCC(k,I) cat E(LP(s_k-1,J),a));
     else {J ∈ Basis(s_k-1); carry forward I's LCC value }
       LCC(k-1,J) := Min(LCC(k-1,J), LCC(k,I))
   end {For all I }
   k := k-1; { Consider next deeper stack state }
  end { While }
 end {LR_Insert}
```

As an example, reconsider grammar G2. We have already exhibited the closure graph for $s_\emptyset$, the start state. The closure graph for s_1, $s_\emptyset$'s successor under '(' is very similar to that of $s_\emptyset$:



Now assume we try to parse '(($'. When the error is detected (assuming the IED property), the parse stack is $(s_\emptyset s_1 s_1)$ and '$' is the error symbol. LR_Insert is then invoked (assume unit costs are used).

(Step 1) {k=3, process the stack top (state s_1) }

The sole basis item is I_1 = [T --> (·E)]. Thus Insert = E('E)',$) = '?'. $LCC(3,I_1)$ = S('E)') = 'a)'.

Since $C(LCC(3,I_1))$ = 2 < C(Insert) = ∞, we continue processing. $Pred(s_1,I_1)$ = [T --> ·(E)] = I_5. I_5 is a closure item.

The sole basis item in state s_1 is I_1. Further from $CG(s_1)$ we can readily observe that $LP(s_1,I_5,I_1)$ = $\{+T\}^*\{)\}$. Thus $LCC(2,I_1)$ = Min(?, $LCC(3,I_1)$ cat $S(LP(s_1,I_5,I_1))$) = Min(?,'a)' cat ')') = 'a))'.

Similary, $LP(s_1,I_5)$ = $\{+T\}^*(\{)\} \cup \{\lambda\})$. Therefore Insert = Min(?, $LCC(3,I_1)$ cat $E(LP(s_1,I_5),\$)$)= Min(?, 'a)' cat ?) = '?'.

(Step 2) { Now k=2 }

Since $C(LCC(2,I_1))$ = 3 < C(Insert) = ∞, we continue processing.

$Pred(s_\emptyset,I_1)$ = [T --> ·(E)] = I_5 which is a closure item. The sole basis item of $s_\emptyset$ is I_1 = [Z --> ·E$].

$LP(s_\emptyset,I_5,I_1)$ = $\{+T\}^*\{\$\}$. Therefore $LCC(1,I_1)$ = Min(?, $LCC(2,I_1)$ cat $S(LP(s_\emptyset,I_5,I_1))$) = Min(?, 'a))' cat '$') = 'a))$'.

$LP(s_\emptyset,I_5)$ = $\{+T\}^*(\{\$\} \cup \{\lambda\})$. Therefore Insert = Min(?, $LCC(2,I_1)$ cat $E(LP(s_\emptyset,I_5),\$)$ = Min(?, 'a))' cat λ) = 'a))'.

Step (3) {k = 1}

Since we have reached the stack bottom, we are done. LR_Insert returns 'a))' as its least cost insertion.

In the above example LR_Insert had to search the entire parse stack to determine a least-cost insertion. However in many cases only a small portion of the parse stack needs to be considered. For example, had the input been '(())$', we would have invoked LR_Insert with the same parse stack. This time though, while processing the stack top, we would get $LCC(3,I_1)$ = S('E)') = 'a)' and Insert = E('E)',')') = 'a'. Since C('a') < C('a)') no cheaper insertion is possible and we would return immediately.

4. A Locally Least-Cost Correction Algorithm

We now add deletion operations to the correction process. We assume a user-supplied deletion cost function where for $a \in \hat{V}_t$, $D(a) \geq 0$ is the cost of deleting 'a'. D($) is fixed at ∞ (because the endmarker is guaranteed to be correct). D can immediately be extended to terminal strings: $D(a_1 \ldots a_m) = D(a_1)+\ldots+D(a_m)$. Assume this correction algorithm is invoked in a situation where $x \in V_t^*$ has already been read (and accepted) by the parser and $b_1 \ldots b_m$ is the remaining input ($m \geq 1$, $b_1, \ldots, b_m \in \hat{V}_t$). That is, $Z' \Rightarrow^* x\ldots$ but $Z' \not\Rightarrow^* xb_1\ldots$ Now a correction is characterized by two parameters, $i \geq 0$, the number of input symbols to delete, and $y \in V_t^*$, the string to be inserted *after* any deletions.

A *locally least-cost correction* is therefore defined as a pair (i,y) which is an optimal solution to the following:

$$\min_{0 \leq i < m,\ y \in V_t^*} \{ D(b_1 \ldots b_i) + C(y) \mid xyb_{i+1}\ldots \in L(G) \}$$

The following routine, which uses LR_Insert as a subroutine, computes locally least-cost corrections for LR(1)-based parsers.


```
function LR_Corrector(s1...sn,b1...bm) : (Del,Insert);
{ s1...sn is the LR-parse stack,
  b1...bm is the remaining input,
  Del is the number of input symbols to delete,
  Insert is the string to insert after all deletions }

Insert := ?; Del := 0;
for I := 1 to m do
  if D(b1...bI-1) >= C(Insert) + D(b1...bDel)
  then { No lower cost correction is possible }
    return;
  if C(LR_Insert(s1...sn,bI)) + D(b1...bI-1)
    < C(Insert) + D(b1...bDel)
  then begin {A better correction has been found }
    Insert := LR_Insert(s1...sn,bI);
    Del := I - 1
  end
 end {For}
end {LR_Corrector}
```

LR_Corrector operates incrementally, first trying 0 deletions, then 1 deletion, etc. This continues until the endmarker (b_m) is reached or until no cheaper correction is possible (because the best known correction is no more expensive than the current cumulative deletion cost). This organization can readily be implemented. As long as no correction of finite cost is known, input symbols already considered (i.e., $b_1, b_2, \ldots$) can be deleted (since there is no correction which will allow them to be accepted). Once a finite cost correction is found (say Del=i, Insert=y), subsequent input symbols must be saved (e.g., in a queue) since they may be needed once parsing is restarted.

At this point we need to continue considering input symbols only to verify that the currect correction is least-cost (or to determine a cheaper one). Normally, only a few more symbols will need to be examined. In particular, we need never look beyond symbol b_j where $D(b_{i+1} \ldots b_j) \geq C(y)$. Since deletion costs are often set rather high (to discourage wholesale deletion of a user's input), once *any* correction is found, we tend to rapidly converge to the locally optimal correction. Indeed, as discussed in section 6, our tests indicate that the costs involved in computing locally optimal corrections are quite reasonable and apparently no real problem in actual production compilers.

5. Formal Properties of the Error-Corrector

We now consider some of the most important formal properties of the correction algorithm introduced above. Implementation and test results are discussed in the next section. We first establish correctness and local optimality -- any input string can be corrected and parsed via a sequence of locally optimal corrections. The following notation will be used:

(1) For a right sentential form $\alpha\beta$, $\alpha \cdot \beta$ will denote this sentential form with α selected as a viable prefix[9].

[9] A viable prefix ([2], p. 380) is a prefix of a right sentential form which does not extend past the handle of that form.

(2) $L(RS)$, where RS is a regular set over $\hat{V}$, is the set of all terminal strings derivable from members of RS.

(3) The *trailing part* of an item $[A \dashrightarrow \alpha \cdot \beta]$ is $\beta \in \hat{V}^*$.

(4) For CFSM state s and $\beta \in \hat{V}^*$, $GOTO(s,\beta)$ is the successor state to s under β.

Definition

Let $S = s_1 \ldots s_n$ be an LR parse stack corresponding to some viable prefix. Assume $I = [A \dashrightarrow \alpha \cdot \beta] \in s_i$ $(1 \leq i \leq n)$. Let $w \in \hat{V}_t^*$ and assume $s_j' = GOTO(s_i, \beta)$. Then w is a *completor* of I in s_i if and only if the parser when restarted with stack S and some input of the form w... can consume w and reach a parse stack configuration $s_1 \ldots s_i \ldots s_j'$ without popping any of $s_1 \ldots s_i$.

□

Recall that informally a completor is simply a string which may be used to complete recognition of some item in a state in the parse stack.

Lemma 5.1

During execution of LR_Insert, if LCC(i,I) contains a string y ≠ ?, then y is a completor for item I in Basis(s_i).

Proof : See Appendix A.1.

□

Lemma 5.2

Assume that after reading and processing some input prefix $y \in V_t^*$ an LR(1)-based parser invokes LR_Insert with an error symbol of 'a'. Then during the execution of LR_Insert, whenever Insert contains a value $z \neq$?, it is the case that $Z' \Rightarrow^+ yza\ldots$.

Proof: See Appendix A.2.

□

Theorem 5.3

Assume that for some LR(1)-based cfg, G, $x\ldots \in L(G)$ but $xa\ldots \notin L(G)$ for $x \in V_t^*$, $a \in \hat{V}_t$. Further, assume that while attempting to parse $xa\ldots$ an LR(1)-based parser invokes LR_Insert as soon as 'a' is encountered. Then LR_Insert will find a least-cost $y \in V_t^+$ such that $Z' \Rightarrow^+ xya\ldots$ if such a string exists. If no such y exists, it will return '?'.

Proof: See Appendix A.3.

□

Theorem 5.4

Assume that some LR(1)-based parser for a cfg, G, is processing an input of $xb_1\ldots b_m$ and that $x\ldots \in L(G)$ but $xb_1\ldots \notin L(G)$ for $x \in V_t^*$ and $b_1,\ldots,b_m \in \hat{V}_t$. Then if LR_Corrector is invoked as soon as b_1 is encountered, it will compute a locally least-cost correction (i,y) $(0 \le i < m,\ y \in V_t^*)$ such that $xyb_{i+1}\ldots \in L(G)$.

Proof:

Follows immediately from the correctness and local optimality of the LR_Insert routine.

□

Corollary 5.5

Let x$ be any input string where $x \in V_t^*$. Then any LR(1)-based parser using LR_Corrector will be able to parse and accept x$.

Proof:

Each invocation of LR_Corrector will return a correction which allows at least one more (non-deleted) input symbol to be accepted by the parser.

□

The above results establish the correctness, local optimality and robustness of the LR_Corrector routine. We now turn our attention to efficiency issues. In considering the space and time requirements of LR_Corrector, it is important to note that the corrector and associated parser will almost certainly not be used in their full generality. In particular, LR(1)-based parsers invariably use a bounded depth parse stack (i.e., a parse stack with a fixed maximum depth). Such parsers accept a string x$ iff x$ $\in L(G)$ and the parse stack does not overflow while processing the input. For modest maximums (e.g., 50 to 100), overflows are so rare that only pathologic inputs are excluded[10]. So too, deletion costs of zero, although allowed by our model, seem never to be used[11] (since they make wholesale deletions far too easy). It is easy to establish that LR_Corrector, when used with a bounded depth parse stack and strictly positive deletion costs, is linear in operation.

<u>Theorem 5.6</u>

Assume a bounded-depth LR(1)-based parser uses LR_Corrector with strictly positive deletion costs. Then an input of x$ will be processed using (a) $O(|x|)$ time and (b) constant space.

<u>Proof</u>: See Appendix A.4.

□

In general, the maximum possible LR parse stack depth grows linearly with the size of the input being parsed. This implies that LR_Corrector, as it stands, can exhibit non-linear behavior. In particular, while parsing an input string x, $O(|x|)$ syntax errors are possible. Each error can require an invocation of LR_Corrector. If zero deletion costs are allowed, all of the remaining input may need to be examined by LR_Corrector, requiring $O(|x|)$ invocations of LR_Insert. Since LR_Insert may need to search the entire parse stack, each invocation of

---

[10]E.g., programs with parentheses or blocks nested deeper than the stack depth limit.

[11]Indeed, some models allow only strictly positive correction costs [5].

LR_Corrector may take $O(|X|^2)$ time with a total of $O(|x|^3)$ time needed to process all of x.

However, as shown in ([8], Theorem 2.5.5), this worst case time bound can be avoided by judicious extensions to LR_Insert and LR_Corrector. LR_Insert can be modified to perform <u>bottom-up</u> rather than <u>top-down</u> parse stack traversals. By saving intermediate information in the parse stack, we can guarantee that a given parse stack state will never be visited more than once for any particular terminal symbol. This allows all invocations of the modified LR_Insert routine to be performed in $O(|x|)$ time. Unfortunately, while this modification yields a better worst case, it yields an inferior <u>average</u> case both because of its extra complexity and because, working in a bottom-up manner, it must normally search the entire parse stack to determine a least-cost insertion.

Deletion costs of zero are handled by preprocessing the input so that when LR_Corrector is invoked, pointers are available to the first occurrence (if any) of each terminal symbol in the remaining input. Obviously if the locally optimal correction is to delete up to a terminal symbol 'b' and then to insert LR_Insert($s_1 \ldots s_n$,b), we need only delete up to the <u>first occurrence</u> of b in the remaining input (to which we have a pointer). This means LR_Corrector needs to only invoke the modified LR_Insert routine $|V_t|$ times and since LR_Insert visits a given stack state at most once for each terminal symbol, an overall linear worst case can be guaranteed. Thus the following can be established.

Theorem 5.7

Let G be any LR(1)-based cfg.

Then there exists a locally least-cost LR(1)-based parser/error-correction algorithm which can correct and parse any input string x$ in O(|x|) time and space.

Proof: See [8], Theorem 3.2.2.

□

6. Implementation and Test Results

The LR_Corrector algorithm has been implemented and tested on a number of LR(1)-based cfgs including LALR(1) grammars for a variant of ALGOL 60 and Pascal. The speed of table generation was quite acceptable, requiring about 5 minutes for the ALGOL grammar ($|V_t|=66$, $|V_n|=81$, $|P|=175$) and 8 minutes for the Pascal grammar ($|V_t|=69$, $|V_n|=130$, $|P|=275$) on a Digital Equipment VAX-11/780. Generation time was about equally divided between the parsing tables and the error correction tables.

Error tables can be generated in many forms, depending on the particular size/speed tradeoff desired. At one extreme, we can precompute and table as much information as possible, seeking to make correction as fast as possible. In testing this approach we tabled the D function as well as the following for each CFSM state s:

(1) For each $I = [A \dashrightarrow \alpha \cdot \beta] \in Basis(s)$:

(a) $S(\beta)$

(b) For each $a \in V_t$: $E(\beta,a)$

(c) For each CFSM state, s': Pred(s',I)

(2) For each $J \in Closure(s)$:

(a) For each $K \in Basis(s)$: S(LP(s,J,K))

(b) For each $a \in V_t$: E(LP(s,J),a)

These tables can be rather large, requiring, e.g., 400K bytes for the ALGOL grammar and 600K bytes for the Pascal grammar. Normally, of course, they would not be kept in main memory, but rather on secondary storage. In such a form, all the information tabled for a state s could be be read into main memory as state s in the parse stack was processed by LR_Corrector. Using this organization, error correction proved to be very fast, requiring an average of about 16 ms. per error (excluding file access time).

As an alternative, we can greatly reduce table sizes by using labelled path information and the S and E functions to compute insertion values as they are needed. To test this approach, we tabled the following:

(1) For each $a \in V_t$: D(a)

(2) For each $X \in V_n$: S(X)

(3) For each $X \in V_n$ and each $a \in V_t$: E(X,a)

(4) For each basis item $[A \dashrightarrow \alpha \cdot \beta]$: β

(5) For each item I and CFSM state s : Pred(s,I)

(6) For each pair of items, I and J in CFSM state s :
RLP(s,J,K) where RLP is a restriction of LP(s,J,K) to paths which do not traverse a given arc more than once[12]

Using this organization, the ALGOL grammar required 56K bytes and the Pascal grammar required 115K bytes. Again, components (3) to (6) can conveniently be stored on secondary storage. In this case, an average correction time of 22 ms. resulted (again excluding file access time).

The difference in correction speeds between the two approaches is surprisingly small. Although one can construct pathological cases in which there is a large difference in speed, in ordinary situations the labelled path information (which the first approach precomputes in detail) is simple enough in structure that direct computation of insertion strings is practical. Because the latter approach requires far less table

[12]Since we are only interested in computing least-cost paths, RLP can clearly be used in place of LP in LR_Insert. Note too that RLP values can be represented as regular expressions using only catenation and alternation.

space and yet is only marginally slower, it appears distinctly preferable.

As mentioned in section 4, tests indicate that very few iterations of LR_Corrector (i.e., calls to LR_Insert) are needed to determine a locally optimal correction. By setting deletion costs to very high values it is possible to estimate the overhead involved in computing locally optimal corrections. When a set of fairly well tuned deletion costs is used rather than uniformly high costs, an increase of only about 50% in the average correction time is observed. This clearly indicates that relatively few deletions, on the average, need be pondered in finding a least-cost correction. Further, those situations where deletions are used are often those in which insertions alone would be long and costly or would lead to later "spurious" errors. Thus the actual costs of finding locally optimal corrections are even lower than the above estimate since such costs are in large part offset by the reduced parsing and correction times they allow.

The following short program provides examples of the kinds of corrections effected by LR_Corrector. The correction costs used are listed in appendix A.5. The original program is first presented using a "↑" to flag symbols considered erroneous. Next, the corrections performed by the algorithm are displayed with insertions underlined with *'s for emphasis and deletions "commented out" with '{' and '}'.

The original program:

```
1.   program ex(input, output);
2.   var a: array [ i : 10 ] of integer;
                        ↑
3.        b: array [ 1..10  2..20 ] ;
                           ↑        ↑
4.        i, j, k, l : integer;
5.  begin
6.    l: i + j > k + l + 4
             ↑
7.    then write ( i ; )
      ↑               ↑ ↑
8.    b ], 2] := 3 * ( i/ + j ;
        ↑↑  ↑ ↑           ↑   ↑
9.    if i = l then then goto l;
                      ↑
10. end end .
        ↑
```

The corrected program:

```
1.  program ex (input, output);
2.  var a: array [ i {:} .. 10 ] of integer;
                         **
3.       b: array [ 1..10 , 2..20 ] of id ;
                          *         *****
4.       i, j, k, l : integer;
5.  begin
6.    l: i := + j > k + l + 4
           **
7.    ; if constant then write ( i ) ; {)}
    ************                  *
8.    b := l {,} + 2 {]} ; id := 3 * ( i / {+} j ) ;
        **         *       ****                   *
9.    if i = l then if constant then goto l;
                    ****************
10. end {end} .
```


Most of the corrections performed in the above example are quite reasonable, but a few point up limitations of our approach. For example, in line 6, '<u>if</u>' should probably be inserted before 'i'. Such a correction <u>cannot</u> be performed by LR_Corrector (or most other correction techniques) because 'i' has already been consumed by the parser when the error is detected. Some correctors ([16], [17]) advocate a "backward move" in such a situation but, as noted in section 2, this can be very difficult in a one-pass compiler since symbols accepted by the parser may already have been translated. An interesting alternative is the use of <u>error productions</u> as discussed in [10]. In this case a production of the form "<If Head> --> λ" can be added to the cfg to <u>anticipate</u> the possible absence of an <u>if</u> header. While too many such productions can make a cfg bulky and unreadable (and possibly even <u>ambiguous</u>), the judicious use of a few such productions seems an excellent way to augment the performance of an error corrector without making wholesale changes to the corrector or its host compiler.

Another problem appears in line 8 in which ...b[1,2]... is probably intended. The difficulty here is that LR_Corrector seeks only <u>local</u> optimality (i.e., a least-cost way of making the first non-deleted input symbol acceptable). In this case, the locally optimal correction (insertion of ':=') leads to later spurious errors. This choice can be avoided if more context is made available (e.g., via a "foward move" phase as suggested by

([16], [17], [31])). However once again this is a very substantial extension to the correction process and it can have undesirable interactions with the rest of the compilation process. An alternate way of viewing the problem is that context-sensitive rules (e.g., type and scope rules) are ignored in the correction process. Thus the correction LR_Corrector chooses is wrong because "b" is an array and may not be assigned an integer value. Indeed, had the input been ...i 1, 2] ..., a forward move scheme might again insert a '[' after the 'i', although in this case context-sensitive rules would bar such a correction.

The problem of using context-sensitive information in the correction process has been studied in [8]. This approach, although as yet untested, seems to have great potential for improving the overall quality of the correction process. In this case error productions again seem to be of real value in enhancing the performance of the LR_Corrector routine. The idea here is to add new symbols and productions to represent <u>some</u> context-sensitive rules. Thus rather than just having a single terminal symbol, 'id', we might have a number of identifiers representing various classes of identifiers (e.g., <array id>, <scalar id>, <procedure id>, etc.). Note that such information can readily be determined by a scanner by merely doing a symbol-table lookup before returning a token to the parser. Now the grammar is modified so that a '[' can follow an <array id>

but <u>not</u> a <scalar id> or <procedure id>. This allows us to lower the cost of inserting a '[' since we have restricted the context in which a '[' may appear. Modifications such as these to the underlying cfg, although fairly straightforward, are extremely useful in enhancing the performance of LR_Corrector at a very modest cost. As another example, note that it is very easy to add another identifier class, <undeclared id>. Deletion costs can be set so that it is much cheaper to delete an <undeclared id> than it is to delete other sorts of identifiers. This allows the correction process to be much more discerning in determining which symbols are to be considered correct and which are to be considered suspect.

It is clear that the behavior of any error-corrector can be considerably altered by changes to the cost functions. The "optimal" selection of insertion costs is, however, a difficult problem, and is usually dealt with in an ad-hoc manner. Two sets of costs used in our experiments are listed in appendices A.5 and A.6; another set may be found in [35]. The interested reader may find a more detailed discussion of cost selection in [34].

To evaluate the performance of our error-corrector, we adopted the criteria of Pennello and DeRemer [31]: a repair is rated "excellent" if it repairs the text as a human reader would, "good" if the repair is not what a human would do but nevertheless is reasonable and introduces no spurious errors, and "poor" if the repair results in one or more spurious errors. By these criteria, LR_Corrector, in the above example, performed 8 excellent corrections, 1 good correction and 2 poor corrections[13].

We compared LR_Corrector with the Simple Precedence corrector of Graham and Rhodes [17], the SLR(1) corrector of Tai [35], and the "insertion-only" LL(1) corrector of [12]. All four techniques were applied to a 63 statement ALGOL program from [34]. The correction costs used by LR_Corrector are listed in appendix A.6.

	Excellent	Good	Poor
LL(1) [12]	45%	26%	29%
SP [17]	40%	42%	18%
SLR(1) [35]	41%	51%	8%
LR_Corrector	61%	25%	14%

The performance of LR_Corrector is quite impressive and is certainly comparable, or superior to, the other correction

[13] Spurious errors are not included in error counts as they are not considered "real".

algorithms. It is important to note however, that the performance criteria used are rather subjective and open to a wide degree of interpretation. Thus, we adjudged a correction poor _whenever_ it led to subsequent "spurious" errors. In cases where a "cluster" of errors appear, however, it is natural for LR_Corrector to sometimes do a correction _incrementally_, with one invocation effecting part of a correction, and subsequent invocations completing the correction. Consider, for example, an error such as ...i := * / i;... One possible correction would be to delete both '*' and '/', which would be rated "good" or even "excellent". LR_Corrector, on the other hand, would correct the error in two steps: first an 'id' would be inserted before the '*', then, on a subsequent invocation, the '/' would be deleted. By our strict interpretation, the first error repair must be deemed poor as it induces a spurious error. But the overall correction obtained, ...i := id * i;... is comparable in quality to ...i := i;... as both require two repair operations. This suggests a slightly weaker definition of a poor correction: a correction is _poor_ if it, and any subsequent corrections it induces, are manifestly inferior to what a human would choose. Thus the correction performed in line 8 of the example is still considered poor because of the large number of unnecessary correction actions it induces. The correction of ...i := * / i... into ...i := id * i;... however, is (more reasonably) rated "good" under our revised definition. Using this revised definition, the performance of LR_Corrector on the

ALGOL test program is now: 61% excellent, 33% good and only 6% poor. These figures seem representative of LR_Corrector's performance on "typical" user programs and certainly suggest that the algorithm's behavior is satisfactory for all but the most demanding of compilers.

It is interesting to compare LR_Corrector's performance with that of the LL(1) correction scheme as both implement the same local least-cost model of correction. The difference in performance between the two is therefore almost wholly[14] attributable to the fact that the LL(1) correction technique performed only insertions (i.e., is essentially an LL(1) version of LR_Insert). The main difference between the two is an increase of about 15% in the number of "poor" corrections attributed to the LL(1) routine. This figure is then an estimate of the fraction of syntax errors which require deletion operations to effect a satisfactory repair. It is a bit surprising that the figure is so low, and it tends to support the conjecture of [12] that an insertion-only corrector can be used in practice with satisfactory results.

[14]The two correctors used somewhat different insertion costs.

7. Conclusion

The error corrector presented has many attractive properties. It presents a very high level correction model in which corrections are determined solely by correction costs and the <u>language</u> being processed. The corrector is usable with any LR(1)-based parsing technique and is automatically generable. The technique can be guaranteed to correctly handle any input and all corrections are locally optimal. In cases of practical interest linearity can easily be established.

Test results are equally encouraging. The corrector has little impact on parsing speed even when processing very ill-formed inputs. Space requirements are acceptable because most of the error tables can be kept on secondary storage. The quality of the error corrections obtained appears to be satisfactory for all but the most demanding of applications.

This correction technique can be used as a basis for further research into more advanced aspects of error correction. The question of how best to assign correction costs for common programming languages needs a great deal of study. So too, ways of extending the limits of this method need to be explored. As described in [10], judiciously chosen error productions seem to be of great value in handling certain difficult cases. Ways of increasing the context available in choosing corrections <u>without</u>

unduly impacting the structure or efficiency of the host compiler are of interest. Also, methods which include <u>context-sensitive</u> considerations (e.g., type and scoping rules) in the correction process, as described in [8], have the potential to greatly enhance overall correction quality and certainly deserve careful study.

In summary, a single definitive and universal correction algorithm for LR(1)-based parsers seems most unlikely to ever emerge. Rather, a hierarchy of techniques, each characterized by its cost, complexity and performance, should be anticipated. Our technique fits nicely into the middleground of such a hierarchy. It is powerful enough to be used in quality compilers but is also simple enough to avoid the costs and complexities of more elaborate schemes. As such, we believe it to be a useful addition to the pantheon of context-free correction techniques.

Acknowledgments

We are grateful to Frank Horn for carefully reviewing earlier versions of this paper.

Appendix

A.1 Proof of Lemma 5.1

By induction on the depth of s_i in the parse stack.

Basis step: s_i is the stack top. Let I = [A --> $\alpha\cdot\beta$]. Then LCC(i,I) = S(β) which is trivially a completor for I in s_i.

Induction step: Assume lemma is true for state s_{i+1}; consider s_i immediately below it in the stack. Again, let I = [A --> $\alpha\cdot\beta$]. Now LCC(i,I) can be assigned a value $\neq$? in one of two ways. If I has an immediate successor item in s_{i+1}, then LCC(i,I) can be assigned the LCC value of that successor (line 30). This LCC value is, by induction, a valid completor for I's successor and hence is also a valid completor for I in s_i. Otherwise, LCC(i,I) can be assigned a value LCC(i+1,K) <u>cat</u> S(LP(s_i,K,I)) (lines 24-25). LCC(i+1,K) is a completor for some closure item K in s_i because it is a completor for a successor of K in s_{i+1}. S(LP(s_i,K,I)) can be written as S(Y_1) <u>cat</u> ... <u>cat</u> S(Y_m) where $Y_1,\ldots,Y_m$ ($m \geq 1$) label a path (K,$J_1,\ldots,J_{m-1}$,I) from K to I in s_i's closure graph. It is easy to verify that LCC(i+1,K) <u>cat</u> S(Y_1) is a completor for item J_1 and thus by a simple induction LCC(i+1,K) <u>cat</u> S($Y_1 \ldots Y_m$) is a completor for item I in s_i.

$\square$

A.2 Proof of Lemma 5.2

Insert is assigned a value $\neq$? in only two places and only when the new value has a cost less than the current value (and thus a cost < C(?) = ∞). In line 11, $E(\beta_I,a)$ can be assigned to Insert when item I = [A --> $\alpha\cdot\beta$] is processed. In this case the desired result follows from the definition of the E function. In line 28, an item J ∈ Closure(s_{k-1}) is considered and Insert can be assigned a value of the form u cat t. String u is the LCC value corresponding to J's successor in s_k. By Lemma 5.1 it is a completor for this successor item and thus also for J. String t = E(LP(s_{k-1},J),a) and can be written as $S(\gamma_1)$ cat ... cat $S(\gamma_{j-1})$ cat $E(\gamma_j,a)$ ($j \geq 1$) where $\gamma_1,\ldots,\gamma_j$ label some path $J,L_1,\ldots,L_j$ in s_{k-1}'s closure graph. String q = LCC(k,I) cat $S(\gamma_1\ldots\gamma_{j-1})$ is (by the arguments of Lemma 5.1) a completor for item L_{j-1} in s_{k-1}. Thus after reading q, the parser can reach a configuration in which the top stack state contains an item [C --> $\rho B\cdot\gamma_j$] where item L_j = [C --> $\rho\cdot B\gamma_j$]. At this point $E(\gamma_j,a)$ cat 'a' can clearly be read by the parser and thus q cat $E(\gamma_j,a)$ = u cat t is a legal value for Insert.

□

A.3 Proof of Theorem 5.3

By Lemma 5.2, we know any string assigned to Insert must be correct and a new value is assigned to Insert only if it is cheaper than the current value. If no least-cost y ∈ V_t^+ exists, '?' must be returned (since the algorithm must halt). We need only therefore show that at some point an attempt to assign a string of cost C(y) must be made. This will be done by showing

how the execution of LR_Insert traces the various ways 'ya' might be recognized once parsing is restarted.

Initial step: It may be that ya... is generated by the trailing part of some basis item [A --> $\alpha\cdot\beta$] in the top stack state. It must be that C(E(β,a)) = C(y) (since y is least-cost) and E(β,a) can then be assigned to Insert at line 11. Otherwise, write ya as y_1y_2a and assume $y_1 \in V_t^*$ is used to complete some item I = [B --> $\gamma\cdot\delta$]. y_1 must be least-cost and thus $C(y_1)$ = C(S(δ)) = C(LCC(i,I)). If C(Insert) > C(LCC(i,I)) = $C(y_1)$, we go on to the next step (otherwise a least-cost solution has already been found).

Iterative step: We have just completed processing a basis item I in state s_j where C(LCC(j,I)) = $C(y_1)$. We continue by tracing how y_2a might be recognized. Item K, I's predecessor in s_{j-1} is considered. It may be the case that y_2a is fully recognized by items in s_{j-1}. If this is so, a sequence of items $K,J_1,\ldots,J_n$ ($n \geq 1$) in CG(s_{j-1}) must exist where segments of y_2a are used to complete, in turn, $J_1,\ldots J_{n-1}$ and the remainder of the string is recognized by the trailing part of J_n. Now it must be the case that C(E(LP(s_{k-1},J),a)) = $C(y_2)$ since LP includes all possible paths from an item and, by assumption, y_2 is least-cost. Thus in line 28 Insert can be assigned a string of cost $C(y_1) + C(y_2)$ = C(y).

Otherwise, write y_2a as z_1z_2a and assume $z_1 \in V_t^*$ is used to complete items in s_{j-1}. A seqence of items $K,J_1,\ldots,J_n$ ($n \geq 0$) will be followed where J_n ∈ Basis(s_{j-1}) and segments of z_1 will be used to complete, in turn, $J_1, \ldots, J_n$. If $n = \emptyset$ then $J_n' = K$ and $LCC(j-1,K)$ can be assigned a string of cost $C(y_1)$ (line 30) and $z_1 = \lambda$. If $n > \emptyset$, then $C(z_1) = C(S(LP(s_{k-1},K,J_n)))$ (since z_1 must be least-cost) and $LCC(j-1,J_n)$ can be assigned (in lines 24-25) a string of cost $C(y_1) + C(z_1)$. In either case, $LCC(j-1,J_n)$ cannot contain a lower cost string since, by Lemma 5.1, this could be used to complete J_n and a lower cost insertion than y would result. If $C(\text{Insert}) > LCC(j-1,J_n) = C(y_1) + C(z_1)$, this step is repeated on the next state down in the parse stack with J_n renamed I, $y_1 z_1$ renamed y_1 and $z_2 a$ renamed $y_2 a$. If $C(\text{Insert}) \leq C(LCC(j-1,J_n))$, the algorithm may terminate but a least-cost Insert value must already have been found since $C(LCC(j-1,J_n)) \leq C(y)$.

The iterative step is repeated until the state which finishes the recognition of ya is processed or until C(Insert) is greater than or equal to the cost of all LCC values. In either case a simple induction on the number of iterative steps executed establishes that an Insert value of cost C(y) must be obtained.

□

A.4 <u>Proof of Theorem 5.6</u>

First note that because the parse stack depth is constant bounded, the result is immediate for the parser itself. Note too that if a reduction stack is needed to guarantee the IED

property, its maximum depth can be constant bounded and thus constant time suffices to reset the parse stack configuration.

Now consider error correction. Each invocation of LR_Insert requires constant time (because each stack state can be processed in constant time). The size of Insert returned by LR_Insert can also be constant bounded (because each stack state contributes a piece of bounded size). Consider each iteration of LR_Corrector as it processes input symbols. Until LR_Insert returns a value $\neq$?, we know the input symbols already considered (b_1, b_2, ...) will <u>have</u> to be deleted. Each of these iterations is charged to the input symbol considered and each such symbol is charged only once. Once LR_Insert returns a value $z \neq$?, we can bound the number of additional iterations needed by C(z). (since each additional iteration represents a possible deletion costing at least one). But, as noted above, the maximum size of z (and thus of C(z)) can be bounded by a constant. Therefore the total time required to find a least-cost correction once <u>any</u> finite cost correction is discovered is constant bounded. This time, as well as the time to insert and later parse the "Insert" string is charged to the first non-deleted input symbol which is guaranteed to be consumed once parsing is restarted.

□

A.5 Pascal Correction Costs

	and	array	begin	case	const	div	downto	else	end
Insertion	5	10	10	10	10	4	8	6	8
Deletion	5	20	20	20	20	4	10	10	15

	file	for	forward	function	goto	if	label	nil	not
Insertion	10	10	10	15	6	10	10	5	5
Deletion	20	20	20	25	10	20	20	15	5

	of	or	packed	procedure	program	record	repeat
Insertion	5	5	10	15	5	10	10
Deletion	5	5	10	25	5	20	20

	set	then	to	type	until	var	while	with	constant
Insertion	10	8	8	10	8	10	10	10	5
Deletion	15	20	12	20	10	20	20	20	15

	identifier	relational op	:=	,	.	..	;	:	↑	+
Insertion	8	4	4	4	7	4	4	7	5	4
Deletion	15	4	9	8	8	8	10	7	6	4

	-	*	/	(	)	[	]	=
Insertion	4	4	4	10	4	10	4	4
Deletion	4	4	4	20	5	20	5	4

A.6 ALGOL 60 Correction Costs

	and	array	begin	boolean	do	end	else	false	for
Insertion	6	11	10	10	8	8	6	7	10
Deletion	6	20	20	20	15	15	10	15	25

	go	label	if	integer	not	or	own	procedure	read
Insertion	9	11	15	10	6	6	10	12	10
Deletion	5	20	25	20	6	6	20	25	20

	real	step	string	switch	then	to	true	until
Insertion	10	8	11	11	6	9	7	8
Deletion	20	12	20	20	10	5	15	12

	value	while	write	identifier	string const
Insertion	11	10	10	8	7
Deletion	20	20	20	15	15

	arith const	relational op	;	+	-	*	/	//	**
Insertion	6	5	5	5	5	5	5	6	5
Deletion	15	5	14	5	5	5	5	5	5

	..	->	==	,	.	:	:=	(	)	[	]
Insertion	4	6	6	4	10	8	9	10	4	10	7
Deletion	10	6	6	8	20	8	9	20	10	20	10

References

1. Aho, A.V., Peterson, T.G. A minimum distance error correcting parser for context-free languages. SIAM Journal of Computing 1,4, 305-312 (1972)

2. Aho, A.V., Ullman, J.D. The Theory of Parsing, Translation and Compiling, Vol. 1, Sec. 5.2. Englewood Cliffs, N.J.: Prentice-Hall 1972

3. Aho, A.V., Ullman, J.D. The Theory of Parsing, Translation and Compiling, Vol. 2, pp. 674-675. Englewood Cliffs, N.J.: Prentice-Hall 1973

4. Aho, A.V., Ullman, J.D. Principles of Compiler Design, pp. 391-402. Reading, Mass.: Addison-Wesley 1977

5. Anderson, S.O., Backhouse, R.C. Least-cost error recovery in LR parsers: a basis. Heriot-Watt University, Edinburgh

6. Conway, R.W., Wilcox, T.R. Design and implementation of a diagnostic compiler for PL/I. Comm. ACM 16, 169-179 (1973)

7. DeRemer, F.L. Simple LR(k) grammars. Comm. ACM 14, 453-460 (1971)

8. Dion, B.A. Locally least-cost error correctors for context-free and context-sensitive parsers. University of Wisconsin-Madison, Ph.D. thesis, Tech. Report 344, December 1978

9. Feyock, S., Lazarus, P. Syntax-directed correction of syntax errors. Software Practice and Experience 6, 207-219 (1976)

10. Fischer, C.N., Mauney, J. On the role of error productions in syntactic error correction. To be submitted to Computer Languages.

11. Fischer, C.N., Milton, D.R. A locally least-cost LL(1) error corrector. To be submitted to IEEE Trans. on Software Engineering.

12. Fischer, C.N., Milton, D.M., Quiring, S.B. Efficient LL(1) error correction and recovery using only insertions. To appear in ACTA Informatica.

13. Fischer, C.N., Tai, K.C., Milton, D.R. Immediate error detection in strong LL(1) parsers. Information Processing Letters 8, 5, 261-266 (1979)

14. Fischer, C.N., Dion, B.A. On testing for insert-correctability in context-free grammars. University of Wisconsin-Madison, Tech. Report 355, July 1979. Submitted to Acta Informatica

15. Ghezzi, C. LL(1) grammars supporting an efficient error handling. Information Processing Letters 3, 6, 174-176 (1975)

16. Graham, S.L., Haley, C.B., Joy, W.N. Practical LR error recovery. Proc. of the Sigplan Sym. on Compiler Construction, in Sigplan Notices, 14, 8, 168-175 (1979)

17. Graham, S.L., Rhodes, S.P. Practical syntactic error recovery. Comm. ACM 18, 639-650 (1975)

18. Gries, D. The use of transition matrices in compiling. Comm. ACM 11, 26-34 (1968)

19. Gries, D. Compiler Construction for Digital Computers, pp. 320-326. New York: Wiley 1971

20. Holt, R.C., Barnard, D.T. Syntax-directed error repair and paragraphing. Submitted to IEEE Trans. on Software Engineering

21. Irons, E.T. An error-correcting parse algorithm. Comm. ACM 6, 669-673 (1963)

22. James, L.R. A syntax directed error recovery method. University of Toronto, M.S. thesis, Computer Systems Research Group Tech. Report CSRG-13, May 1972.

23. LaFrance, J.E. Syntax directed error recovery for compilers. University of Illinois, Ph.D. thesis, Illiac IV Doc. 249, 1971

24. Leinius, R.P. Error detection and recovery for syntax directed compiler systems. University of Wisconsin-Madison, Ph.D. thesis, 1970

25. Levy, J.P. Automatic correction of syntax errors in programming languages. Cornell University, Ph.D. thesis, Tech. Report TR 71-116, 1971

26. Lewi, J., DeVlaminck, K., Huens, J., Huybrechts, M. The ELL(1) parser generator and the error recovery machanism. Acta Informatica 10, 209-228 (1978)

27. Lewis, P.M., Rosenkrantz, D.J., Stearns, R.E. Compiler Design Theory, pp. 276-284, 462-469, 526-531. Reading, Mass.: Addison-Wesley 1973

28. Lyon, G. Syntax-directed least-errors analysis for context-free languages: a practical approach. Comm. ACM 17, 3-14 (1974)

29. Mickunas, M.D., Modry, J.A. Automatic error recovery for LR parsers. Comm. ACM 21, 459-465 (1978)

30. Pai, A.B., Kieburtz, R.B. Global context recovery: a new strategy for parser recovery from syntax errors. Proc. of the Sigplan Sym. on Compiler Construction, in Sigplan Notices, 14, 8, 158-167 (1979)

31. Pennello, T.J., DeRemer, F.L. A forward move algorithm for LR error recovery. Proc. Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, pp. 241-254, 1978

32. Peterson, T.G. Syntax error detection, correction and recovery in parsers. Stevens Institute of Technology, Ph.D. thesis, 1972

33. Poplawski, D.A. Error recovery for extended LL-Regular parsers. Purdue University, Ph.D. thesis, August 1978

34. Rhodes, S.P. Practical syntactic error recovery for programming languages. University of California at Berkeley, Ph.D. thesis, Tech. Report 15, 1973

35. Tai, K.C. Syntactic error correction in programming languages. IEEE Trans on Software Engineering Vol. SE-4, 5, 414-425 (1978)

36. Teitlebaum, R. Context-free error analysis by evaluation of algebraic power series. Proc. ACM SIGACT Fifth Annual Conference on Theory of Computing, Austin, Texas, pp. 196-199, 1973