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

AN INSERTION ONLY ERROR CORRECTOR FOR LR(1),LALR(1),SLR(1) PARSERS

by

Bernard A. Dion
and
Charles N. Fischer

Computer Sciences Technical Report # 315

February 1978

AN INSERTION ONLY ERROR CORRECTOR FOR LR(1),LALR(1),SLR(1) PARSERS

by

Bernard A. Dion

and

Charles N. Fischer

Computer Sciences Technical Report #315

February 1978

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. It chooses least-cost insertions (as defined by the user) in correcting syntax errors. Moreover, the error-corrector can be generated automatically from the grammar and a table of terminal symbol insertion costs. An algorithm that decides if a given LR(1) grammar is insert-correctable is presented. Correctness and linearity of the error-corrector is established.

1. INTRODUCTION

The problem of error recovery and correction in bottom-up context-free parsing has received much attention. Among others, James [9] presents a "phrase level recovery" which allows the restart of a parser while skipping a minimal amount of input; Graham and Rhodes [7] try to bring context information on the parsing stack by using a "forward move/ backward move" in a simple precedence parse; Druseikis and Ripley [5] extend the same ideas to SLR(1) parsing. However all of these techniques, when faced with certain syntax errors, are forced to skip ahead in the input stream, completly ignoring portions of it.

We consider an error corrector which generates "locally least cost corrections" in the presence of any syntax errors working with LR(1) parsers and variations such as SLR(1) and LALR(1). Following [6], we restrict our attention to insert-correctable grammars, that is those grammars for which it is always possible to effect a correction by insertion of a terminal string.

2. DEFINITIONS AND NOTATIONS

Let $G = (V_n, V_t, P, S)$ be a context-free grammar. We will use the corresponding augmented grammar :

$$G' = (V_n \cup \{S'\}, V_t \cup \{\$\}, P \cup \{S' \rightarrow S\$\}, S')$$

All input string will be terminated by the end marker symbol $\$ \notin V_t$. Further, we assume $S' \notin V_n$. Let us denote $V_t \cup \{\$\}$ by $\hat{V}_t$ and $V_n \cup \{S'\}$ by $\hat{V}_n$. Also let $\hat{V} = \hat{V}_t \cup \hat{V}_n$.

Given an input string $xa\ldots$ ($x \in V_t^*$, $a \in \hat{V}_t$) such that $S' \Rightarrow^+ x\ldots$ but $S' \not\Rightarrow^+ xa\ldots$, the correction algorithm will find $y \in V_t^+$ such that $S' \Rightarrow^+ xya\ldots$. In order for this algorithm to succeed, we need to guarantee two properties [6]. One is related to the class of languages considered, the other is related to the parsing algorithm that is used :

(a) A cfg is said to be insert-correctable iff for all $x \in V_t^*$ and $a \in \hat{V}_t$ such that $S' \Rightarrow^+ x\ldots$ and $S' \not\Rightarrow^+ xa\ldots$ there exists $y \in V_t^+$ such that $S' \Rightarrow^+ xya\ldots$ [6] .

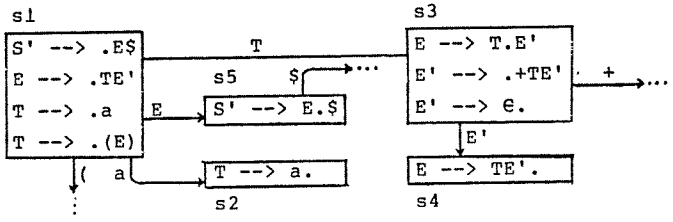
(section 6 presents an algorithm that decides if an LR(1) grammar is insert-correctable)

(b) In order to have our error corrector working properly , we need to detect an error upon first encountering the erroneous symbol [6]. This will be achieved when using a canonical LR(1) parser [3]. However this is not true when using a standard SLR(1) or LALR(1) parser. Because of the use of approximations to exact lookaheads [4], it is possible to do some incorrect reductions when using an erroneous symbol as the lookahead.

Example 2.1 : Consider the following insert-correctable grammar G1

```
S' --> E $
E  --> T E'
E' --> + T E' | ϵ
T  --> a | ( E )
```

The following is part of G1's CFSM:



Now try to parse 'a)$'. The parsing sequence is :

Step	Lookahead	Stack	Output	Accepted Program Prefix
1	a	s1	scan	a
2	)	s1 s2	reduce{T->a}	T
3	)	s1 s3	reduce{E'->ϵ}	TE'
4	)	s1 s3 s4	reduce{E->TE'}	E
5	)	s1 s5	error	E

It is now too late for our algorithm to do a correction; in fact at this point no insertion at all is possible. Note that the parser made an erroneous transition in step 3 because) ϵ FOLLOW(E').

In order to solve the above problem, we propose the following modification to standard SLR(1) or LALR(1) parsers :

Algorithm 2.1 Parser Configuration Recovery Routine

> Whenever a new symbol $l \in \nabla_t$ is used as the lookahead, we initialize an auxiliary stack buffering the parser moves as they occur. If a syntax error is detected while l is the lookahead, we restore the parser stack to the state it had when l became the lookahead symbol, using information kept in the auxiliary stack. If l is accepted by the parser (i.e., scanned), we clear the auxiliary stack for the next lookahead symbol.

For the previous example, a "restore" operation in step 5 yields the following :

Step	Lookahead	Parser Stack	Auxiliary Stack
5	)	s1 s5	reduce{E'->ϵ}, reduce{E->TE'}
2'	)	s1 s3	empty

We are now ready to do a correction. The next two sections will show how the appropriate insertion (in this case '+(a') can be determined.

3. RIGHT CONTEXT OF AN ITEM IN A PARSER STATE

In order to be able to do a least cost insertion to the immediate left of an error symbol we need to know which strings can appear in this position. Note that this same problem was easier in the LL(1) error corrector described in [6] since, in this case, what we expect to see is stored in the parsing stack.

In the following sections we combine the terminologies of Aho and Ullman [3] and De Remer [4]. The theory that is developed applies in the same way to LR(1), LALR(1) and SLR(1) parsers.

We will partition items of a state s of the CFSM in the following way [4] :

$$s = Basis(s) \cup Closure(s)$$
$$Basis(s) = \{ I=[A \rightarrow \alpha.\beta] \mid I \in s, \alpha \neq \epsilon \}$$
$$Closure(s) = \{ I=[B \rightarrow .\gamma] \text{ or } I=[C \rightarrow \epsilon.] \mid I \in s \}$$

with the exception that [S' --> .S$] in s_1 (the initial state) is considered a basis item.

We know that an LR(1) parser generates a rightmost derivation in reverse (i.e., a rightmost parse) [3]. The theory necessary to construct LR(1) 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.

Definition 3.1

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) $S' \Rightarrow^* \alpha\beta$
(2) There does not exist γ such that $S' \Rightarrow^* \alpha\gamma$ and $\gamma \Rightarrow^+ \beta$

Informally (2) says that no further reduction can apply to β (i.e., β is fully reduced). Note that we need the fact that LR(1) grammars are unambiguous [8] in order to guarantee the existance of a viable suffix for a given viable prefix. For certain (ambiguous) cfg's and sentential forms no viable suffix exists (e.g. S --> aA$, A --> a | B, B --> A and $S \Rightarrow^* a\ldots$).

Definition 3.2

The right context $R(I,s_1s_2\ldots s_n)$ of item $I=[A \rightarrow \beta_1.\beta_2]$ where the contents of the parse stack are $S = s_1s_2\ldots s_n$ and $I \in s_n$, is the set of strings $\beta_2\gamma \in \Sigma^*$ such that $\beta_2\gamma$ is a viable suffix of G' corresponding to $\alpha\beta_1$ (i.e., $S' \Rightarrow_{rm}^* \alpha Aw \Rightarrow_{rm} \alpha\beta_1\beta_2 w$) where

(1) $\alpha\beta_1 = X_1X_2\ldots X_{n-1}$ $\quad X_k \in \Sigma$ for $k = 1,\ldots,n-1$

(2) $GOTO(s_k,X_k) = s_{k+1}$ $\quad$ for $k = 1,\ldots,n-1$
where GOTO is the usual transition function as in [3].

$S' \Longrightarrow^*_{rm} \alpha Aw \Longrightarrow_{rm} \alpha\beta_1\beta_2 w$ says that $I = [A \rightarrow \beta_1 . \beta_2]$ is a valid item for $\alpha\beta_1$ [3]. Note that for some $\beta_2 Y \in R(I,S)$ it is (by definition) the case that $\beta_2 Y \Longrightarrow^* \beta_2 w$. Conditions (1) and (2) simply say that the part of the input string that has already been scanned reduces to $\alpha\beta_1$ and yields a parsing stack $S = s_1 s_2 \ldots s_n$.

We now consider the problem of computing R(I,S). We will show that it can be expressed as a regular expression over ∇. R(I,S) will be written as the concatenation of 3 regular expressions over ∇ :

$$R(I,S) = \beta_2 \ \underline{\text{cat}} \ l(I,s_n) \ \underline{\text{cat}} \ g(I,S)$$

where cat denotes concatenation

$\beta_2 \in \nabla^*$ is the string following '.' in I

l is a regular expression over ∇ denoting the local right context .

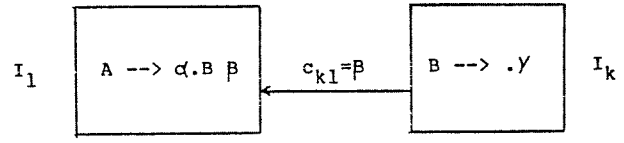
g is a regular expression over ∇ denoting the global right context .

We now explain what is meant by local and global right context.

(a) The local right context $l(I,s_n)$ of item I in state s_n is that part of R(I,S) which can be obtained by considering only the predictions done in computing the closure set of state s_n. That is, $l(I,s_n)$ is that part of R(I,S) determined solely by s_n (independently of the rest of S).

Suppose $[A \rightarrow \alpha . B\beta] \in s_n$ (where $B \in \nabla_n$) , then for all $B \rightarrow \gamma \in P$ we add item $[B \rightarrow .\gamma]$ to the closure set

of s_n as described in [4]. We may also build a closure graph $G(s_n)$ whose nodes are items in s_n. If we obtain item $I_k = [B \rightarrow .\gamma]$ from item $I_l = [A \rightarrow \alpha . B\beta]$, we put an edge (I_k, I_l) in $G(s_n)$ and label it with $c_{kl} = \beta$. β is that part of the local right context of I_k that we bring from I_l by doing a prediction :



Example 3.1 : Consider the following insert-correctable SLR(1) grammar which will be used in all the examples that follow.

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

Figure 3.1 shows part of G2's CFSM (s1 and s2) and G(s1), the closure graph of state s1.

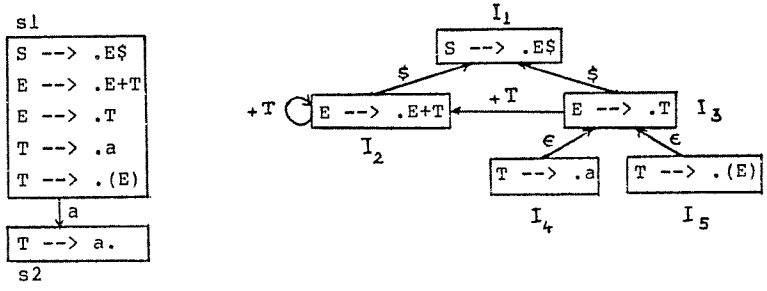


figure 3.1

Let $\{ I_k , k = 1 \text{ to } |s| \}$ be the set of items in s. We want to compute $l(I_k,s)$, the local right context of item I_k in state s. In order to do this we need to consider all paths in G(s) between I_k and any item in the basis of s. Each time an edge (i,j) is used on a path, c_{ij} is concatenated to the local right context. The following algorithm, similar to the "all paths" algorithm[1] given in [1], computes $l(I_k,s)$, for all k.

Algorithm 3.1

```
Begin
   For i,j from 1 to |s| do
      l^∅_ij <-- if there exists (i,j) ∈ G(s)
                 then {c_ij} else ∅;
   For i from 1 to |s| do
      l^∅_ii <-- l^∅_ii U {ϵ}
   For k from 1 to |s| do
      For i,j from 1 to |s| do
         l^k_ij <-- l^(k-1)_ij U (l^(k-1)_ik cat (l^(k-1)_kk)* cat l^(k-1)_kj) ;
   For i from 1 to |s| do
      l(I_i,s) <-- U l^|s|_ij
                   j s.t. {I_j ∈ Basis(s)}
End
```

In the above example, we obtain the following regular expressions:

$$l(I_1,s_1) = \{\epsilon\}$$
$$l(I_2,s_1)=l(I_3,s_1)=l(I_4,s_1)=l(I_5,s_1)=\{+T\}^* \text{ cat } \{\$\}$$

---

(1) cf [1] p.198; we are using
+ = union $\emptyset = \emptyset$
. = concatenation $1 = \{\epsilon\}$

(b) The global right context g(I,S) for item I and parsing stack $S = s_1 s_2 \ldots s_n$ is that part of R(I,S) which is to the right of $l(I, s_n)$. It is termed global because it is determined by all of S and not just s_n. Unlike the local right context, it cannot be determined at the time the parser is generated. We will show that g(I,S) can be computed as a function of local right contexts of items in states $s_{n-1}, s_{n-2}, \ldots, s_1$.

Assume we want to compute $g([A \text{ --> } X.\beta], s_1 s_2 \ldots s_n)$. Note that this implies we are dealing with a basis item (the general case will be treated in the next section). We have to consider the predecessor of item $[A \text{ --> } X.\beta]$ in state s_{n-1} as shown in figure 3.2 .

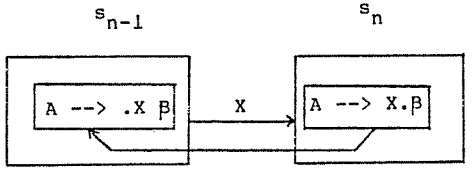


figure 3.2

We can write :

$$g([A \text{-->} X.\beta], S) = l([A \text{-->} .X\beta], s_{n-1}) \text{ cat } ([A \text{-->} .X\beta], s_1 s_2 \ldots s_{n-1})$$

That is, global right context is obtained by concatenating appropriate local right contexts for each state of the parsing stack.

Example 3.2

Reconsider example 3.1. Now suppose we want to compute the right context of item I_1 = [T --> a.] in state s_2 assuming the parse stack is $s_1 s_2$.

We have $\beta_2 = \epsilon$ and $l(I_1, s_2) = \{\epsilon\}$ since $I_1 \in Basis(s_2)$. Now I_1 is obtained from [T --> .a] in state s_1 by scan operation and we have $l([T \text{ --> } .a], s_1) = \{+T\}^*$ cat $\{\$\}$. Moreover $g([T \text{ --> } .a], s_1) = \{\epsilon\}$ (this is because s_1 is at the bottom of the parsing stack). So that we get

$$g(I_1, s_1 s_2) = \{+T\}^* \text{ cat } \{\$\}$$

At this point we know how to compute local right contexts and we could present an algorithm to compute global right contexts. However we really don't need to compute full expressions for right contexts when doing insertions to the left of an error symbol. Rather, we just need to consider those strings that have a chance to be used in a least cost insertion (non least cost insertions are of no interest for our purposes).

4. THE ERROR CORRECTOR

As in [6], we will assume a given insertion cost vector C with $C(\epsilon) = 0$, $C(\$) = 1000$ and $C(a) \geq 0$ supplied for all $a \in V_t$. Note that C($) is assigned a large value because $ can never be inserted during error correction -- it is always correctly provided as an "end of input" flag. We intentionally avoid setting C($) to infinity or the largest representable integer to ensure that the concept of a least cost string is always well defined (e.g., C(a$)>C($)) and that overflow will not occur during least cost computations. We also introduce a special symbol '?' $\notin \bar{V}$ such that C(?) = infinity.

We can formulate our error correction problem as an optimization 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 the following problem :

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

Note that we are looking for a locally optimal solution in the sense that we may not find the best of all possible corrections. The latter approach was taken in [2] and seems to lead to impractical solutions in the sense that they are too expensive with respect to time and space bounds (in example 2.1, assuming that all terminal insertion costs are set to 1, we change 'a)$' into 'a+(a)$', a better correction would be found by the algorithm given in [2] i.e. changing 'a)$' into '(a)$').

Also note that our insertions are done in such a way that we never modify the string of symbols that has been accepted by the parser so that a compiler may use them for semantic and code generation purposes. A more global approach would certainly require more work in this respect.

Our error correction algorithm requires the computation of error correction tables. This computation is naturally embedded in the parser generator and the tables are later used by the error corrector which is embedded in the parser itself.

We first give two preliminary definitions :

(a) For $\alpha \in \Diamond^*$, define $S(\alpha)$ to be an optimal solution to the following problem:

$$\min \{ C(y) \mid \alpha \Longrightarrow^* y \}$$
$$y \in \Diamond_t^*$$

(b) For $A \in \Diamond_n \cup V_t$ and $a \in V_t$, define Prefix(A,a) to be an optimal solution to the following problem:

$$\min \{ C(y) \mid A \in \Diamond, A \Longrightarrow^* ya\ldots \}$$
$$y \in \Diamond_t^*$$

Algorithms for computing $S(\alpha)$ and Prefix(A,a) are given in [6] and need not be reproduced here (Prefix corresponds to E in the reference cited).

Generation of Error Correction Tables :

(a) Let $I_k = [A \rightarrow \alpha.\beta\ ] \in$ Basis(s) :

i. If the error symbol $a \in \Diamond_t$ is not generable from β we may need to insert the least cost string derivable from β, so we need to table $S(\beta)$.

ii. The least cost insertion which will allow $a \in \Diamond_t$ to be generated by this item where $\beta = U_1U_2\ldots U_n (n \geq \emptyset)$ is $S(U_1U_2\ldots U_i)$ cat Prefix(U_{i+1},a) where i minimizes

$$\min_{\emptyset \leq i \leq n-1} C(S(U_1\ldots U_i) \text{ cat } \text{Prefix}(U_{i+1},a))$$

Call this string Insert(β,a). If $\beta \neq\Longrightarrow^* \ldots a \ldots$ then Insert(β,a) = ? .

iii. In the event no optimal insertion can be generated from state s we will have to generate insertions based upon state s', an immediate predecessor of s. Therefore we need to table a list of possible predecessors of I_k . Elements of this list will be pairs (1,s') such that state s' is an immediate predecessor of s in the CFSM and $I_1 \in$ s' is the item which produced I_k by a scan operation. Call this list Pred(I_k) and note that it is trivially obtained when building the CFSM.

(b) Let $I_k = [A \rightarrow .\gamma\ ] \in$ Closure(s)

i. For such an item we need to table a list of backpointers to all items in Basis(s) that can be reached from it in G(s). Each backpointer is a pair (1,y) where $I_1 \in$ Basis(s) and y is a least cost terminal string that can be used to reach I_1 from I_k. Call this list $B(I_k)$.

In example 3.1, we have $B(I_5)$ = {(1,\$)} meaning basis item I_1 is the only one that can be reached from $I_5 \in G(s_1)$ and the least cost terminal string that can be used along a path from I_5 to I_1 is "\$".

$B(I_k)$ for all k such that $I_k \in$ Closure(s) can be obtained by using a shortest path algorithm on G(s) where the cost of using edge (i,j) is $C(S(c_{ij}))$. Note that this shortest path algorithm is a special case of the all paths algorithm given in the previous section and can be found in [1].

ii. We also need to table the least cost insertions that can be used to generate the error symbol locally (meaning using local right context only). This is a solution to the following minimization problem:

$$\min_{y \in V_t^*} \{ C(y) \mid \delta ==>^* ya... \text{ and } \delta \in 1(I_k,s) \}$$

Call these strings $T(I_k,a)$ where $a \in \nabla_t$ is the error symbol. $T(I_k,a)$ = "?" means no insertion possible (which is why C(?) = infinity).

Example 4.1

Using example 3.1 we have $T(I_4,')') = '+(a'$ using path (I_4,I_3,I_2), getting $\in$ from (I_4,I_3) and '+(a' as +.Prefix(T,')') from (I_3,I_2).

Algorithm 4.1

$T(I_k,a)$ for all $I_k \in$ Closure(s) and all $a \in \nabla_t$ can be obtained in the following way :

$T(I_k,a)$ <-- ?
For l,m from 1 to |s| do
  Comment let s_{kl} be a least cost terminal string obtained when following a path from I_k to I_l (from min path algorithm) , if one exists, otherwise '?'. Further c_{lm} = '?' if no arc from l to m exists ;
If $C(s_{kl}$ cat $Insert(c_{lm},a)) < C(T(I_k,a))$
  Then $T(I_k,a)$ <-- s_{kl} cat $Insert(c_{lm},a)$;

The structure of the error correction tables can be summarized in the following way :

For all states s $\in$ CFSM
[1] For all k such that $I_k \in$ Basis(s)
  Comment Assume $I_k = [A_k \text{ --> } \alpha_k.\beta_k]$;
  Table (i) $Insert(\beta_k,a)$ for all $a \in \nabla_t$
    (ii) $S(\beta_k)$ a least cost terminal string derivable from β_k
    (iii) $Pred(I_k)$ the list of predecessors of I_k

[2] For all k such that $I_k \in$ Closure(s)
  Comment Assume $I_k = [A_k \text{ --> } .\gamma_k]$;
  Table (i) $B(I_k)$ the list of backpointers to items in Basis(s)
    (ii) $T(I_k,a)$ for all $a \in \nabla_t$ a table of insertions using the local right context of I_k

Error Corrector Procedure

Now we are ready to give the error corrector procedure. This procedure would be used with a standard LR(1) parser or a modified SLR(1)/LALR(1) parser (as described in section 2). It is called upon detection of a syntax error. Its input is a $\in \nabla_t$, the error symbol and implicitly the parse stack. It computes INSERTION $\in \nabla_t^+$, the desired locally least cost insertion string.

Before we exhibit the procedure, let us introduce the notion of an error correction graph : let $S = s_1 s_2 \ldots s_p$ be the parsing stack at the time the error symbol is first seen as the lookahead. We will process the stack in a top-down fashion in order to determine INSERTION. As we visit the states on the parsing stack, we create stages (figure 4.2). A stage has the error correction table of the corresponding state s associated with it. It also has one node labeled LC_i for each item $I_i \in$ Basis(s). The contents of LC_i is a string in $\nabla_t^* \cup \{?\}$.

If we are dealing with the initial stage (corresponding to s_p) , the LC_i's are the $S(\beta_i)$'s where $I_i = [A_i \text{--}> \alpha_i.\beta_i]$. In general LC_i is a least cost terminal string that can be used to the right of the last accepted symbol if item I_i is to be used (i.e. valid) during the parse of the string to be inserted. LC_i is maintained to cover the possibility that the error symbol will be generated by a predecessor of I_i (in a deeper state in the stack).

Consider figure 4.1; item $I_1 \in$ Basis(s_k) is linked to $I_m \in s_{k-1}$, the item which produced I_1 by scan operation. This item I_m is known to be uniquely determined by I_1 and s_{k-1}. It is given to us by Pred(I_1).

Now I_m is linked to $I_{m_1}, I_{m_2}, \ldots, I_{m_t} \in$ Basis(s_{k-1}) (where $t=t(m)\geq 1$). These are basis items of s_{k-1} reachable from I_m such that $(I_m,\ldots,I_{m_1}),\ldots,(I_m,\ldots,I_{m_t})$ are minimal cost paths in $G(s_{k-1})$. They are given to us by the back-pointer list $B(I_m)$.

Assume $B(I_m) = \{(m_1,y_1),\ldots,(m_t,y_t)\}$. A possible value for LC_{m_i} (in the stage corresponding to s_{k-1}) is LC_1 cat y_i. This follows from the fact we know LC_1 is the lowest cost insertion necessary to make I_m valid once parsing is restarted. Further, we know y_i is the lowest cost terminal string which links I_m to I_{m_i} (that is, which makes I_{m_i} valid given that I_m is). We therefore assign LC_1 cat y_i to LC_{m_i} if no lower cost insertion string is known (which might make I_{m_i} valid through a different closure item).

This calculation of LC values corresponds to lines 16-29 of algorithm 4.2 . Note that if $I_m \in$ Basis(s_{k-1}) we just transfer the LC_1 contents to LC_m if no lower cost insertion string is already known for LC_m. (lines 37-39 of algorithm).

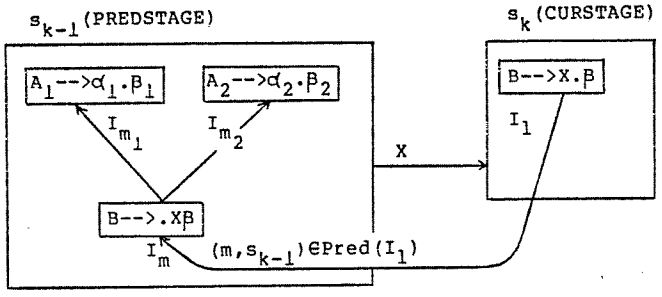
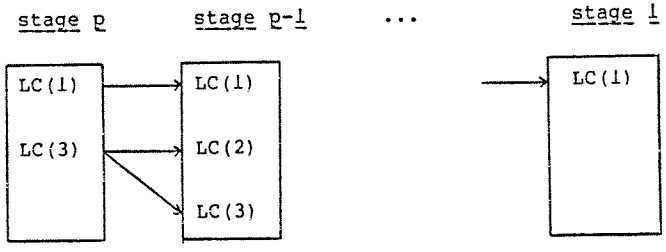


figure 4.1

As we process the parsing stack we don't need to keep track of all stages at any given time, just the current stage CURSTAGE and the previous stage PREDSTAGE. STAGE(s) is a function which returns a pointer to a new stage corresponding to state s and initializes all LC_i's to "?". The following figure, 4.2, shows how stages of an error correction graph are processed :



error correction graph
figure 4.2

We also want to keep track of the least cost INSERTION that can be obtained given that the error symbol is generated by local context in a state already examined. Starting with state s_p, we initialize INSERTION to an optimal solution of the following problem :

$$\min \{C(\text{Insert}(\beta_i,a)) \mid I_i=[A_i \text{ --> } \alpha_i.\beta_i]\}$$
$$I_i \in \text{Basis}(s_p)$$

This corresponds to lines 5-10 of algorithm 4.2 .

Now consider figure 4.1; if $I_1 \in s_k$ is linked back to $I_m \in \text{Closure}(s_{k-1})$, we want to set INSERTION equal to the string which minimizes :

$$\min \{C(\text{INSERTION}), C(\ LC_1 \ \underline{\text{cat}}\ T(I_m,a)\ ) \mid (m,s_{k-1}) \in \text{Pred}(I_1) \text{ and } I_m \in \text{Closure}(s_{k-1})\}$$
$$I_1 \in \text{Basis}(s_k)$$

That is, we consider the possibility of obtaining a lower cost value for INSERTION by allowing the error symbol, a, to be generated from the local right context of I_m in s_{k-1} (that is, $1(I_m,s_{k-1})$). $T(I_m,a)$ yields the lowest cost insertion needed to generate a from I_m's local context; LC_1 is the lowest cost insertion possible which will make I_m valid. This computation corresponds to lines 30-35 of the algorithm.

In general we do not need to process the stack down to stage 1. If we are processing stage k and we have $C(\text{INSERTION}) \leq C(LC_i)$ for all i's we know that INSERTION has the optimality property we are looking for. This follows from the fact $C(T(I_m,a)) \geq 0$; hence the termination condition in the while loop on lines 13-14 of the algorithm.

Algorithm 4.2

```
1  Procedure ERRORCORRECTOR(a,INSERTION) ;
2  Begin Comment initialize error correction
3                using top-state info       ;
4     k := p ; INSERTION := ?  ; CURSTAGE := STAGE(s_p) ;
5     For all i such that [ A_i --> α_i.β_i] ∈ Basis(s_p) do
6     Begin
7        CURSTAGE.LC_i := S(β_i) ;
8        If C(Insert(β_i, a)) < C(INSERTION)
9           then INSERTION := Insert(β_i,a)
10    End for all i ;
11    Comment now process stack until no
12           lower cost insertion is possible  ;
13    While there exists i such that
14          C(CURSTAGE.LC_i)  < C(INSERTION) and k > 1 do
15    Begin PREDSTAGE := STAGE(s_k-1) ;
16       For all I_1 ∈ Basis(s_k)  such that
17              C(CURSTAGE.LC_1)  < C(INSERTION) do
18       Begin Comment link I_1  to predecessors
19                    in Basis(s_k-1) ;
20          Let m be such that (m,s_k-1)  ∈ Pred(I_1) ;
21          If I_m ∈ Closure(s_k-1)
22             Then Begin
23                  Comment follow backpointer
24                         to basis items  ;
25                  For all (m_i,y_i) ∈ B(I_m) do
26                     If C(CURSTAGE.LC_1)  + C(y_i)
27                         < C(PREDSTAGE.LC_m_i)
28                     Then PREDSTAGE.LC_m_i
29                              := CURSTAGE.LC_1 cat y_i ;
```



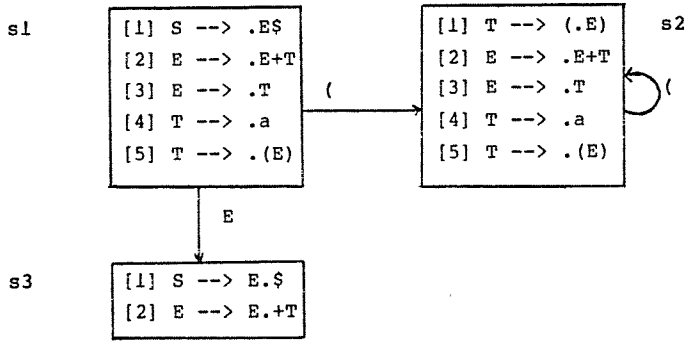
```
30                  Comment if lower cost insertion can be
31                         obtained, update INSERTION  ;
32                  If C(CURSTAGE.LC_1) + C(T(I_m,a))
33                     < C(INSERTION)
34                     Then
35                        INSERTION := CURSTAGE.LC_1 cat T(I_m,a)
36                  End If I_m ...
37              Else Comment we have I_m ∈ Basis(s_k-1)  ;
38                 If C(CURSTAGE.LC_1) < C(PREDSTAGE.LC_m)
39                    Then PREDSTAGE.LC_m := CURSTAGE.LC_1
40       End For I_1 ;
41       CURSTAGE := PREDSTAGE ; k := k-1
42    End While loop
43 End Errorcorrector.
```

Note that the above ERRORCORRECTOR procedure can be used in the case G' is not insert-correctable, In this case we may return from ERRORCORRECTOR with INSERTION = ? (meaning no possible insertion) and have to announce failure (or invoke a heuristic routine).

Example 4.2

We reconsider grammar G2 given in example 3.1

(a) Part of the CFSM is :



(b) Assuming all terminal insertion costs are set to one, we get the following error correction tables (the notation a:y where $a \in \Sigma_t$ and $y \in \Sigma_t^*$ means 'insert y to the left of a').

state s1

basis item I_1

Pred(I_1) = $\emptyset$ (because s1 is the initial state)

Insert('E$') = [$:a,a:ϵ,+:a,(:ϵ,):(a]

S('E$') = 'a$'

closure items I_2,I_3,I_4,I_5

$B(I_2) = B(I_3) = B(I_4) = B(I_5) = \{(1,\$)\}$

$T(I_2) = T(I_3) = T(I_4) = T(I_5) =$
[$:ϵ,a:+,+:ϵ,(:+,):+(a]

(since $1(I_k,s_1) = \{+T\}^*$ cat [$}, k = 2,3,4,5)

state s2

basis item I_1

Pred(I_1) = {(5,1),(5,2)}

Insert('E)') = [$:'?',a:ϵ,+:a,(:ϵ,):a]

S('E)') = 'a)'

closure items I_2,I_3,I_4,I_5

$B(I_2) = B(I_3) = B(I_4) = B(I_5) = \{(1,')')\}$

$T(I_2) = T(I_3) = T(I_4) = T(I_5) =$
[$:'?',a:+,+:ϵ,(:+,):ϵ]

(since $1(I_k,s_2) = \{+T\}^*$ cat {)}, k = 2,3,4,5)

state s3

basis item I_1

Pred(I_1) = {(1,1)}

Insert($) = [$:ϵ,a:?,+:?,(:?,):?]

S($) = '$'

basis item I_2

Pred(I_2) = {(2,1)}

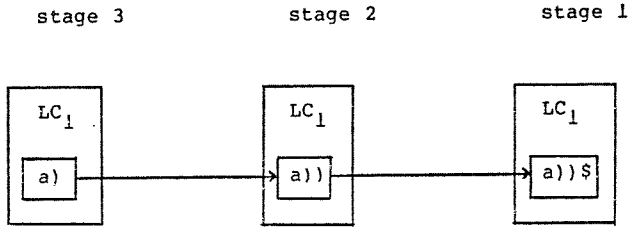
Insert(+T) = [$:?,a:+,+:ϵ,(:+,):+(a]

S(+T) = '+a'

(c) Assume we now try to parse '(($'. We detect a syntax error in the following parsing configuration :

stack : $s_1s_2s_2$; error symbol : $

We obtain the following error correction graph



This is obtained in the following way

(a) Create stage 3

LC_1 <-- S('E)') = 'a)' using [T --> (.E)] in Basis(s_2).

INSERTION = ? since E) =/=>* ...$....

(b) Create stage 2

We have (5,2) $\in$ Pred(I_1) in state 2 and B(I_5) = {(1,')')} in state 2 so that we link LC_1 in stage 3 to LC_1 in stage 2 and set LC_1 in stage 2 equal to 'a))'. No other path exists. There is no local correction that can be obtained from I_5 in stage 2 since T(I_5,$) = ? in state 2.

(c) Create stage 1

We have (5,1) $\in$ Pred(I_1) in state 2 and B(I_5) = {(1,$)} in state 1 so that we link LC_1 in stage 2 to LC_1 in stage 1 and set LC_1 in stage 1 equal to 'a))$'. No other possibility can be found. Also T(I_5,$) = $\in$ in state 1, so that we obtain a correction by concatenating 'a))' and $\in$.

INSERTION <-- 'a))'

We now exit the while loop because we have reached the bottom of the stack and we finally have corrected '(($' into '((a))$'.

The reader may readily verify that (as noted above) an input of 'a)$' would be corrected to 'a+(a)$'. It is important to observe that in many cases least cost corrections can be determined simply (and quickly) from considerations of only the top state on the parse stack. For example, assume an input of 'aa$'. When an error is detected, we are in the following parse configuration :

stack : $s_1 s_3$; error symbol : a

Considering s_3 we obtain LC_1 = S($) = $ and LC_2 = S(+T) = +a. Further, INSERTION = Insert(+T,a) = '+' . Since C(+) is less than both C(LC_1) and C(LC_2), the computation immediately terminates with a correction of 'aa$' into 'a+a$'. The error corrector thus attempts to effect corrections using local con-

text only. When necessary, however, it considers just enough global context to guarantee that the lowest cost correction possible is effected.

5. PROPERTIES OF THE ERROR-CORRECTOR

We now consider some of the most important properties of the parsing algorithm introduced above. We first prove correctness -- any input string can be corrected and parsed. Let us introduce some new notations.

(1) For a right sentential form $\alpha\beta$, $\alpha.\beta$ denotes this sentential form with α selected as a viable prefix (of $\alpha\beta$).
(2) $l(I,J,s)$ denotes the regular set of all paths from item I to item J in G(s) (it may be $\emptyset$).
(3) $L(l(I,s))$ is the set of all terminal strings derivable from members of the regular set denoted by $l(I,s)$.
(4) The trailing part of an item $[A \rightarrow \alpha.\beta]$ is $\beta \in V^*$.

Definition 5.1

Let $S = s_1 \ldots s_n$ be a parse stack corresponding to some viable prefix. Assume $I = [A \rightarrow \alpha.\beta] \in s_i$ $(1 \leq i \leq n)$. Then $w \in V_t^*$ is a completor of I in s_i if and only if the parser when restarted with some input of the form w... , can consume w and reach a parse stack configuration $s_1 \ldots s_i \ldots s_j'$ where $s_j' = GOTO(s_i,\beta)$.

Informally, a completor can be used to complete the recognition of some item in a state in the parse stack.

Lemma 5.1

During execution of Algorithm 4.2, if LC_j in a stage corresponding to state s_i contains a string $\neq ?$ then LC_j is a completor for basis item I_j in s_i.

Proof : may be found in Appendix A.1.

Lemma 5.2

Assume that after reading and processing some input prefix $y \in V_t^*$ an LR parser invokes Algorithm 4.2 with an error symbol of 'a'. During the execution of Algorithm 4.2, wherever INSERTION contains any string $z \neq ?$, it is the case that $S' \Rightarrow^+ yza\ldots$.

Proof : may be found in Appendix A.2.

Theorem 5.1

Assume that for some insert-correctable cfg, G, $x \ldots \in L(G)$ but $xa\ldots \notin L(G)$ for $x \in V_t^*$, $a \in V_t$. Further assume that while attempting to parse xa... an LR parser invokes Algorithm 4.2 as soon as 'a' is encountered. Then Algorithm 4.2 will find and insert $y \in V_t^+$ such that y is an optimal solution to

$$\min_{y \in V_t^+} \{ C(y) \mid S' \Rightarrow^+ xya\ldots \}$$

Proof : may be found in Appendix A.3.

We now analyse the efficiency of our error correcting parser. We first present a quadratic upper bound and later show how Algorithm 4.2 can be modified in order to guarantee linearity. The following lemma and theorem are valid for SLR(1) and LALR(1) parsers as well as LR(1) parsers (since we use Algorithm 2.1 to 'undo' any incorrect parser moves caused by the error symbol). We shall term this class LR(1)-based parsers.

Lemma 5.3

Assume an LR(1)-based parser using Algorithm 4.2 as an error corrector processes x$ and corrects it to x'$. Then $|x'| = O(|x|)$.

Proof : may be found in Appendix A.4.

Theorem 5.2 :

Assume an LR(1)-based parser using Algorithm 4.2 as an error corrector processes x$. Then it requires at most $O(|x|^2)$ time.

Proof : may be found in Appendix A.5.

Although we have established a quadratic upper bound, we believe that for common programming languages and common syntax errors, this quadratic behavior will not be realized. This is because, in most cases, errors will be corrected locally (that is the error symbol will be generated from a state near the top of the parse stack). Another reason which strongly suggests that the average behavior of our error correcting parser will be linear is the fact that LR-driven compilers almost invariably use a fixed size parse stack with a relatively small maximum height (e.g., 100) and yet are able to compile very large programs (e.g., more than 10,000 tokens) without parse stack overflow.

However, if linearity is desired, we can modify Algorithm 4.2 in order to achieve this objective. Note that Algorithm 4.2 performs a top-down stack traversal in order to compute the optimal insertion from left to right. We can easily construct an algorithm performing a bottom-up stack traversal in order to compute the optimal insertion from right to left (this Algorithm is given in Appendix A.7). As is shown below, this algorithm can be very easily made linear. However, its use is not recommended because, in a bottom-up stack traversal, all stack states have to be examined before the optimal insertion can be known, thus giving an average case behavior almost certainly worse than Algorithm 4.2. On the other hand we can combine top-down and bottom-up stack traversals in order to obtain good average and worst case behavior.

This mixed strategy is now outlined. We mark each state in the parse stack with

(a) a count of the number of times this state has been visited during a top-down stack traversal. This count will be kept less than M where $M \geq 1$ is a fixed constant.

(b) a boolean flag for each terminal symbol 'a' indicating whether or not this state has been visisted during bottom-up stack traversal for a correction corresponding to error symbol 'a'.

When a state is pushed, we set the above count to 0 and all flags to false. When an error is detected, we attempt top-down traversal. As we visit a state, we increment its count by 1 only if this count was previously less than M. If we hit a state with count equal to M, we abandon top-down traversal and initiate bottom-up traversal starting at the deepest stack state with the flag corresponding to the error symbol equal to false (we can readily maintain a pointer to the uppermost state for which the flag corresponding to a given error symbol is true). To allow bottom-up traversal to be restarted at this point, we assume that each time the flag is set true for a given error symbol, appropriate pointers to the current IS values (see Algorithm A.7) are saved. Since these values are determined solely by the error symbol and the states below the point of restart, error processing can clearly be restarted at this point. As we visit states, we set the flag corresponding to the error symbol to true and table IS values for the appropriate error symbol.

Theorem 5.3

Assume an LR(1) error correcting parser using the mixed strategy described above processes x$. Then it requires at most O(|x|) time and O(|x|) space.

Proof : may be found in Appendix A.6.

We note that Theorem 5.3 is limited to LR(1) parsers because, in the worst case, Algorithm 2.1 can be non-linear. However in the case of typical programming languages we have very strong reasons to believe this algorithm will require at most a number of moves bounded by a rather small constant. Certainly a parse stack with a small maximum height will guarantee this. Equally important, it is right recursion in a production (direct or indirect) which makes Algorithm 2.1 at times require more than a constant number of moves. However, right recursion is almost invariably avoided in LR parsers (in favor of left recursion) precisely because it increases the parse stack depth required to parse various constructs.

In summary, Algorithm 4.2 appears to be a simple and efficient basis for the automatic correction of a large class of LR grammars encountered in practice. Further, if worst case linearity is required the suggested modifications may be utilized without an excessive degradation of average case performance.

6. INSERT-CORRECTABILITY OF LR(1) GRAMMARS

We now consider the problem of deciding if a given augmented LR(1) grammar G is insert-correctable. This is done by extending the definition of the CFSM corresponding to G. While building the extended CFSM, we consider items of the form I = [A --> $\beta_1.\beta_2$, u, t] where

(i) u $\in \nabla_t$ is the usual LR(1) lookahead symbol. Since u is not relevant to our problem, it will be ignored in the following discussion.

(ii) for any a $\in \nabla_t$ and for any viable prefix $\alpha\beta_1$ for which I (in state s) is valid t(a) = true if and only if I (in s) is valid for some right-most sentential form $\alpha\beta_1.\beta_2 w$ where w = ...a... . Call this condition (*).

We will present an algorithm that computes the extended CFSM, followed by a lemma showing that the t-functions that are computed satisfy condition (*).

Algorithm 6.1 Extended CFSM Computation

[1] The basis of the initial state is $s_\emptyset$ = {[S' --> .S$, t]} where t(a) = false for all a $\in \nabla_t$.

[2] The basis of the X-successor s' of state s is obtained as follows :
For all items [A --> $\beta_1.X\beta_2$, t] in s, add item [A --> $\beta_1 X.\beta_2$, t] to Basis(s').

[3] For a closure item I = [A --> .Y, t_1] in state s :
t_1(a) = true if and only if
(a) t(a) = true for any basis item I' in s which is a descendant of I in the closure graph of s.

or (b) ...a... ∈ L(1(I,s)) .

Lemma 6.1

(a) Let I = [A --> $\beta_1.B\beta_2$,t] be a basis item in state s and let $\alpha\beta_1.B\beta_2 w$ be any right sentential form for which I (in s) is valid. If J = [C --> .γ, t'] is a closure item in s then J is valid for $\alpha\beta_1.\gamma vw$ where v ∈ L(1(J,I,s)).

(b) Let J = [C --> .γ, t] be a closure item in state s and assume J (in s) is valid for $\alpha.\gamma z$. Then there exists a basis item I = [A --> $\beta_1.B\beta_2$, t'] in s such that I is valid for $\delta\beta_1.B\beta_2 w$ where $\delta\beta_1 = \alpha$ and z = vw for some v ∈ L(1(J,s)).

Proof : may be found in Appendix A.8.

Lemma 6.2

If condition (*) holds for all basis items of a state s, it holds for all closure items.

Proof : may be found in Appendix A.9.

Lemma 6.3

Assume we construct an extended CFSM as defined by Algorithm 6.1. Then condition (*) holds for every item in every state.

Proof : may be found in Appendix A.10.

Definition 6.1

A state s of the extended CFSM corresponding to an augmented LR(1) grammar G is safe if and only if for all a ∈ ∇_t there exists a basis item I = [A --> $\beta_1.\beta_2$, t] in s such that $\beta_2 \Rightarrow^* ...a...$ or t(a) = true.

Theorem 6.1

An augmented LR(1) grammar G is insert-correctable if and only if all states of the extended CFSM corresponding to G are safe.

Proof : may be found in Appendix A.11.

7. CONCLUSIONS AND DIRECTIONS FOR FUTURE RESEARCH

We are currently working on the implementatiion of an LALR(1) error correcting parser using Algorithm 4.2 as an error-corrector. Preliminary experience with it suggests that it can operate satisfactorily with most LR-driven compilers. Although the error correction table is quite large, it can be stored efficiently in secondary storage (we differentiate between error-correction tables which are only used on an exception basis and parsing tables which need to be available throughout the compilation process).

This research can be extended in several ways. We plan to include deletions as well as insertions in our error-correction scheme. A correction operation would be effected as follows: deletion of zero or more input symbols followed by insertion of a terminal string. It can be noted that our definition of least-cost correction is a very local one since it is concerned with finding the least-cost insertion which allows the next input symbol to be accepted. We believe it is worhtwhile to develop more global error correction techniques. Recent work by Penello and DeRemer [10] shows how a forward move can be used by an LR 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.

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 on top of the stack. Let I_j = [A --> α.β]. Then $LC_j = S(\beta)$ which is trivially a completor for I_j.

Induction step: assume Lemma true for state s_{i+1}; consider s_i immediatly below it in the stack. Again let I_j = [A --> α.β]. Now LC_j can be assigned a value in one of two ways. If I_j has an immediate successor in s_{i+1} then LC_j is assigned the LC value of the successor (line 39). Since this LC value is a completor for I_j's successor, it must also be a completor for I_j. Otherwise, LC_j is assigned a value LC_n cat y (lines 28-29). LC_n is a completor for a closure item I_k in s_i (because it is a completor for I_k's successor in s_{i+1}) and y is derived from $\gamma \in l(I_k, I_j, s_i)$. y can be written as $y_1 \ldots y_m$ and γ as $\gamma_1 \ldots \gamma_m$ where $\gamma_1, \gamma_2, \ldots$ are the labels (c_{ij} values) on a path $I_k, I_1, \ldots I_j$ in $G(s_i)$ and $\gamma_1 \rightarrow^* y_1, \ldots, \gamma_m \rightarrow^* y_m$. It is easy to verify that LC_k cat y_1 is a completor for I_1 and thus by a trivial induction that LC_k cat y is a completor for I_j.

A.2 Proof of Lemma 5.2

INSERTION is assigned a value 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 9, Insert(β,a) is assigned to INSERTION if the top of the stack state contains an item [A --> α.β]. In this case the desired result follows from the definition of Insert. In line 35, an item $I_m \in$ Closure(s_{k-1}) is considered and INSERTION is assigned a value of the form u cat t. u is the LC value corresponding to I_m's successor in s_k. By Lemma 5.1 it is a completor for this item and thus

also for I_m. t is equal to $T(I_m,a)$ and may be written as t_1t_2. t_1 is derived from a path of length $\geq \emptyset$ from I_m to some item I_n = [B --> .γ] in $G(s_{k-1})$. t_2 is equal to Insert(δ,a) where I_p = [C --> ρ.Bδ] is an immediate successor of I_n. By an induction on path length it can be established that u cat t_1 is a completor for I_n. Thus after reading u cat t_1 the parser can reach a configuration in which the top stack state contains an item [C --> ρB.δ] and t_2a can clearly be read from this configuration.

A.3 Proof of Theorem 5.1

Since G is insert-correctable, some least-cost insertion string y must exist. By Lemma 5.2 we know any string assigned to INSERTION is correct and a new value is assigned to INSERTION only if it is of lower cost than the current value. 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 showing how the execution of Algorithm 4.2 traces the various ways ya may be recognized once parsing is restarted.

Initial step: it may be that ya... is generated by the trailing part of some basis item [A --> α.β] in the top stack state. It must be that C(Insert(β,a)) = C(y) (since y is least cost) and Insert(β,a) is assigned to INSERTION in this case (line 9). Otherwise, write y as y_1y_2a and assume $y_1 \in V_t^*$ is used to complete some basis item I_i = [B --> γ.δ]. y_1 must be least cost and thus $C(y_1)$ = C(S(δ)) = $C(LC_i)$. If C(INSERTION) > $C(LC_i)$ = $C(y_1)$ 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_i in state s_j where $C(LC_i) = C(y_1)$. We continue by tracing how y_2a might be recognized. I_k, I_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 I_k, I_{m_1}, ..., I_{m_n} ($n \geq 1$) in $G(s_{j-1})$ must exist where segments of y_2a are used to complete in turn, I_{m_1}, ..., $I_{m_{n-1}}$ and the remainder of the string is recognized by the trailing part of I_{m_n}. Now it must be the case that $C(T(I_k,a)) = C(y_2)$ since computation of T considers all possible paths from an item and, by assumption, y_2 is least cost. Thus in line 35 INSERTION 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 sequence of items I_k, I_{m_1}, ..., I_{m_n} ($n \geq \emptyset$) will be followed where $I_{m_n} \in$ Basis(s_{j-1}) and segments of z_1 will be used to complete, in turn I_{m_1}, ..., I_{m_n}. If $n = \emptyset$ then LC_{m_n} is assigned a string of cost $C(y_1)$ (line 39) and $z_1 = \epsilon$. If $n > \emptyset$ then $C(z_1) = C(v)$ where $(m_n,v) \in B(I_k)$ (since z_1 must be least cost) and LC_{m_n} is assigned (in lines 28-29) a string of cost $C(y_1)+C(z_1)$. In either case LC_{m_n} cannot contain a lower cost string since, by Lemma 5.1, this could be used to complete I_{m_n} and a lower cost insertion than y would result. If C(INSERTION) > $C(LC_{m_n})$ = $C(y_1)+C(z_1)$ this step is repeated on the next state down in the parse

stack with I_{m_n} renamed I_1, y_1z_1 renamed y_1 and z_2a renamed y_2a. If C(INSERTION) $\leq$ $C(LC_{m_n})$ the algorithm may terminate but a least-cost INSERTION must already have been found since $C(LC_{m_n}) \leq C(y)$.

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

A.4 Proof of Lemma 5.3

We need only show that each symbol inserted during error correction can be charged to some input symbol and that each input symbol is charged for at most a constant number of insertions.

For charging purposes we associate each state with an input symbol. Assume that during normal parsing (when the lookahead symbol is in x$), the stack height is h when symbol 'a' is first used as a lookahead. Any states added by 'a' at a height greater than h are charged to a; those at height $\leq$ h retain the association in effect when 'a' was first used. It is easy to establish that the number of states so charged to 'a' will be bounded by a constant and will not increase as parsing progresses.

Now assume Algorithm 4.2 is invoked with error symbol 'b' and a stack $S = s_1 \ldots s_j$. Starting with s_j, states are visited until at state s_i ($i \leq j$), INSERTION is determined (the fact that Algorithm 4.2 may have to do some processing further down the stack to verify optimality of this correction is of no interest in this proof). INSERTION can be written as LC cat LOCAL where LC is determined by $s_{i+1} \ldots s_j$ and LOCAL is determined by s_i (and of course 'b'). The portion of LC contributed by each of s_{i+1} to s_j can be bounded in length by a constant and is charged to the input symbol associated with each such state.

By construction, LC is a completor (see Definition 5.1) for some closure item [A --> .α] in s_i (if $i<j$) and after LC is fully parsed, the A-successor to s_i is the stack top. Then LOCAL is processed. Its length can be bounded by a constant and it is charged to 'b'. After LOCAL is parsed and just before normal parsing is resumed with 'b' as the lookahead, the number of states above s_i in the stack can be bounded by a constant (determined by s_i, A and LOCAL). Each of these states (created during error processing) is charged to 'b'.

We now observe that the total number of states charged to a given input symbol and used to contribute a portion of LC is

bounded by a constant and thus so is the total number of LC symbols charged to that symbol. So too, an input symbol is charged at most once for LOCAL(which is of bounded length). The desired result is immediate.

A.5 Proof of Theorem 5.2

(1) Assume first an LR(1) parser is used. It is easy to establish that given a careful implementation of Algorithm 4.2, the time required to process each stack state during correction can be bounded by a constant. By Lemma 5.3, at most $O(|x|)$ states can be processed during any invocation of the corrector and no more than $|x|$ invocations are possible. The $O(|x|^2)$ time bound follows immediately.

(2) In the case an SLR(1) or LALR(1) parser is used, we use Algorithm 2.1 to restore the parse stack to the configuration existent when the error symbol was first used as lookahead. The number of parser moves to be undone is bounded (by Lemma 5.3) by $O(|x|)$ and since at most $|x|$ invocations of Algorithm 2.1 are possible at most $O(|x|^2)$ time is added to the total excution time of Algorithm 4.2.

A.6 Proof of Theorem 5.3

(1) Each state is visited at most $M+|V_t|$ times and by careful implementation each visit can be performed in constant time. Using Lemma 5.3, linear time follows.

(2) Linear space can be obtained if we link pieces of LC's, IS's and INSERTION's as they are built. Each state again contributing pieces bounded in size by a constant depending solely on the grammar and by Lemma 5.3 linear space follows. We assume that space is reclaimed when LC, IS or INSERTION values are discarded (this obviously does not change the linear time bound of (1)).

A.7 Bottom-up Stack Traversal LR Error Corrector

The following algorithm computes a string INSERTION having the same properties as that of Algorithm 4.2. However it computes INSERTION from left to right while examining the parse stack in a bottom-up fashion.

```
 1 Procedure ERRORCORRECTOR(a,INSERTION) ;
 2 Begin
 3    CURSTAGE := STAGE(s_1) ;
 4    For all i such that [A_i -->α_i.β_i] ∈ Basis(s_1) do
 5       CURSTAGE.IS_i := '?' ;
 6    p := depth of parse stack ;
 7    For k := 2 to p do
 8    Begin
 9       PREDSTAGE := CURSTAGE ; CURSTAGE := STAGE(s_k) ;
10       For all l such that I_l ∈ Basis(s_k) do
11       Begin Comment link I_l to predecessors
12                     in Basis(s_k-1) ;
13          CURSTAGE.IS_l := '?' ;
14          Let m be such that (m,s_k-1) ∈ Pred(I_l) ;
15          If I_m ∈ Closure(s_k-1) then
16             Begin
17             Comment follow backpointers to basis items ;
18             For all (m_i,y_i) ∈ B(I_m) do
19                If C(y_i cat PREDSTAGE.IS_m_i)
20                   < C(CURSTAGE.IS_l)
21                   then CURSTAGE.IS_l
22                            := y_i cat PREDSTAGE.IS_m_i ;
23             Comment check if local insertion
24                     can be obtained ;
25             If C(T(I_m,a)) < C(CURSTAGE.IS_l) then
26                CURSTAGE.IS_l := T(I_m,a) ;
27             End
```

```
28          Else Comment we have I_m ∈ Basis(s_k-1) ;
29             If C(CURSTAGE.IS_1) > C(PREDSTAGE.IS_m)
30                Then CURSTAGE.IS_1 := PREDSTAGE .IS_m ;
31       End For I_1 ;
32    End For k;
33    INSERTION := '?' ;
34    For all i such that [A_i --> α_i.β_i] ∈ Basis(s_p) do
35       Set INSERTION to an optimal solution to
36          min { C(INSERTION), C(Insert(β_i,a)),
37                                C(s(β_i) cat IS_i) }
38 End ERRORCORRECTOR ;
```

Correctness of this algorithm can be obtained in the same way as that of Algorithm 4.2. Simply notice that IS_i in a stage corresponding to stack state s_j is a least cost terminal string that can be used to the left of the error symbol 'a' if item $I_i \in Basis(s_j)$ is to be used during the parse of the string to be inserted. Also note that this algorithm uses the same tables as Algorithm 4.2.

A.8 Proof of Lemma 6.1

(a) A simple induction on path length from I to J in the closure graph of state s.

(b) A simple induction on path length from J to I in the closure graph using the observation that if closure item J is valid for a right sentential form, this sentential form must have a predecessor in the derivation for which some immediate successor of J in the closure graph is valid.

A.9 Proof of Lemma 6.2

Consider closure item J = [C --> .δ,t], any $a \in V_t$ and any viable prefix γ such that J (in s) is valid for γ.

If t(a) = true, then by construction of Algorithm 6.1 step [3] either (a) or (b) must hold. If (a) holds, let the

the basis item I' be [A --> $\beta_1.B\beta_2$,t_1] where $\gamma = \alpha\beta_1$. By (*), since $t_1(a)$ = true, I' is valid for $\alpha\beta_1.B\beta_2 w$ where w = ... a... . But by Lemma 6.1 (a), we know J is valid for $\alpha\beta_1.\delta vw$ as required. If (b) holds, let I' be as above and assume v = ...a... $\in L(1(J,I',s))$. Then for all $\alpha\beta_1.B\beta_2 w$ for which I' is valid, we know (by Lemma 6.1 (a)) that J is valid for $\alpha\beta_1.\delta vw$ = $\gamma.\delta vw$ as required by (*).

If J is valid for $\gamma.\delta z$ where z = ...a... then by Lemma 6.1 (b), for some basis item I = [A --> $\beta_1.B\beta_2$,t_1], it is the case that I is valid for $\alpha\beta_1.B\beta_2 w$ where $\alpha\beta_1 = \gamma$ and z = vw for some $v \in L(1(J,s))$. If w = ...a... then (since (*) holds for I), $t_1(a)$ = true and thus t(a) = true by construction. Otherwise ...a... = v $\in L(1(J,s))$ and again, by construction, t(a) = true.

A.10 Proof of Lemma 6.3

An induction on the order in which states are created : (*) trivially holds for the sole basis item of $s_\emptyset$ and by Lemma 6.2 (*) then holds for all of $s_\emptyset$. In like manner (*) holds for the basis items of any state that is added because it holds for the (already existing) items from which the basis items were created (by a successor operation). By Lemma 6.2, (*) then holds for all items of the newly created state.

A.11 Proof of Theorem 6.1

(If part) Assume x has been read and reduced to viable prefix γ. Assume further we are in state s. Since s is safe, for any $a \in \nabla_t$ there exists a basis item I = [A --> $\beta_1.\beta_2$,t] for which $\beta_2 \Longrightarrow^* ya...$ or t(a) = true. Clearly for the former case $S' \Longrightarrow^+ xya...$. In the later case, by (*) $S' \Longrightarrow^+_{rm} \alpha\beta_1\beta_2 w$ where w = ...a... and $\gamma = \alpha\beta_1$. Thus $S' \Longrightarrow^+ x...w$ = x...a... .

(Only if part) Now assume s is not safe and assume the items in s are valid for δ where $\delta \Rightarrow^* x \in V_t^*$. Choose any basis item $I = [A \rightarrow \beta_1.\beta_2, t]$ in state s. Since s is not safe $\beta_2 \not\Rightarrow^* \ldots a \ldots$. Further by (*) no sentential form $\alpha\beta_1.\beta_2 w$ in which $w = \ldots a \ldots$ can exist (otherwise t(a) = true). Thus item I cannot participate in any parse which will allow 'a' to be accepted. But neither can any other basis item, so G cannot be insert-correctable.

References

[1] Aho,A.V. Hopcroft,J.E. and Ullman,J.D.
The Design and Analysis of Computer Algorithms
Addison Wesley. Reading, MA 1976. Section 5.6 pp 195-217.

[2] Aho,A.V. and Peterson,T.G.
A Minimum Distance Error Correcting Parser for Context Free Languages
SIAM J.Comp. Vol.1 number 4. December 1972 pp 305-312.

[3] Aho, A.V. and Ullman,J.D.
The theory of Parsing, Translating and Compiling.
Volume I : Parsing
Prentice Hall. Englewood Cliffs, N.J. 1972. Section 5.2 pp 368-396.

[4] De Remer, F.L.
Simple LR(k) Grammars
Communications of ACM. July 1971. Vol.14 number 7 pp 453-460.

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

[6] Fischer, C.N. Milton, D.R. and Quiring, S.B.
An Efficient Insertion-Only Error Corrector for LL(1) Parsers
Conference Record Fourth ACM Symposium on Principles of Programming Languages January 1977. Submitted to ACTA Informatica.

[7] Graham, S.L. and Rhodes, S.P.
Practical Syntax Error Recovery
Communications of ACM. November 1975. Vol.18 Number 11 pp 639-650.

[8] Hopcroft, J.E. and Ullman, J.D.
Formal Languages and their Relation to Automata
Addison Wesley. Reading, MA 1969. Chapter 12 pp 166-188.

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

[10] Penello,T.J. and DeRemer, F.
A Forward Move For LR Error Recovery
Conference Record Fifth ACM Symposium on Principles of Programming Languages January 1978.