A LOCALLY LEAST-COST

LL(1) ERROR CORRECTOR

by

C.N. Fischer, J. Mauney, D.R. Milton

Computer Science Technical Report #371

October 1979

A Locally Least-Cost

LL(1) Error Corrector

C. N. Fischer[1,2]

J. Mauney[1]

D. R. Milton[3]

---

[1]Present address: University of Wisconsin - Madison, Madison, Wi. 53706

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

[3]Present address: Bell Laboratories, Naperville, Il. 60540

Abstract

An LL(1)-based error correction algorithm is presented. The algorithm can be used with any LL(1) grammar and 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. The algorithm is seen to be very fast and quite modest in its primary memory requirements. Further, its performance on test cases is very encouraging.

List of Footnotes

[1]Present address: University of Wisconsin - Madison, Madison, Wi. 53706

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

[3]Present address: Bell Laboratories, Naperville, Il. 60540

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

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

[6]Spurious errors, induced by an error corrector, are not included in error counts as they are not considered "real" user errors.

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

Index Terms

Error correction, error recovery, LL(1) parsing, compilation, least-cost corrections, syntax errors

1. Introduction

In [6], an LL(1) error correction algorithm which operates by "insertion-only" is studied. The algorithm is particularly simple in structure and lends itself to compact and efficient implementations. Further, the corrector can be automatically generated from an LL(1) grammar and a table of terminal insertion costs. The corrections chosen by the algorithm can be shown to be always locally optimal (i.e., locally least-cost). The algorithm is also noteworthy in that it presents a very high level correction model in which all repairs are determined <u>solely</u> by the <u>language</u> to be processed and the insertion costs used (and <u>not</u> by the underlying context-free grammar or details of the correction algorithm). Such a high level model makes using (and, via costs, tuning) the corrector a very simple matter.

Nevertheless, this corrector has two distinct liabilities. First, the corrector cannot be used with all LL(1) grammars, but rather only with a subset termed "insert-correctable." As discussed in [6], most modern programming languages, as typified by Pascal and Algol 60, are <u>not</u> insert-correctable in structure (although they are <u>very</u> close). This means that if the language being parsed is fixed and unalterable, the insertion-only corrector may not be able to repair all errors (i.e., in some situations it will report failure).

Secondly, even in those cases where some insertion can repair a given error, it is obvious that sometimes a simple deletion will be preferable to a possibly long (and costly) insertion. Thus if high quality corrections are desired, deletions must sometimes be allowed.

In this paper we extend the insertion-only LL(1) corrector to include deletion operations. In doing so we create an algorithm which can be used with <u>any</u> LL(1) grammar. Further, the extended corrector's performance is always as good as, and is usually superior to, that of the original algorithm. The extra costs incurred are moderate enough to allow the extended algorithm to be used in production compilers without significantly impacting either their size or speed.

This paper is organized as follows. In section 2 we briefly review the structure and properties of the original insertion-only algorithm. The extended algorithm is presented in section 3, while implementation and test results are discussed in section 4. Finally, in section 5 the results of this work are summarized and possible future research is considered.

2. A Least-Cost Insertion-Only Corrector

In our presentation, we shall assume that the reader is familiar with the basic notions of grammars and parsing [1]. The empty (or null) string is denoted by λ. cat denotes string catenation.

The algorithm presented in [6] is an "insertion-only" error corrector for LL(1) parsers. Because the corrector may need to consider insertions at the end of an input string, it is necessary to use an augmented grammar. Let $G = (V_n, V_t, P, Z)$. Then the augmented grammar $G' = (V_n \uplus \{Z'\}, V_t \uplus \{\$\}, P \cup \{Z' \rightarrow Z\$\}, Z')$, where $\$ \notin V_t$, $Z' \notin V_n$. All input strings will be terminated by the endmarker symbol, \$. We shall consider all grammars 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 as input a string xa... ($x \in V_t^*$, $a \in \hat{V}_t$) such that $Z' \Longrightarrow^+ x\ldots$ but $Z' \not\Longrightarrow^+ xa\ldots$, the correction algorithm will find a least-cost string $y \in V_t^+$ such that $Z' \Longrightarrow^+ xya\ldots$ if such a string exists. Otherwise, it will return a symbol '?' $\notin \hat{V}_t$ as an indication of failure.

The LL(1) parsing algorithm used with the corrector must be constrained. It is well-known that every LL(1) grammar is strong [1], and the conventional parsing algorithm for LL(1) languages

takes advantage of this fact. However, this algorithm will not necessarily detect an error upon first encountering an erroneous symbol ([1], [6]). That is, it does not possess the <u>immediate error detection</u> (IED) property. A simple and efficient way of obtaining the IED property in Strong LL(1) parsers is discussed in [7]. In what follows, we shall assume that the LL(1) parser used possesses the IED property.

The error-correcting algorithm will require two auxiliary tables, S and E. These tables 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 $w = X_1 \ldots X_m \in \hat{V}_t^*$, $C(w) = C(X_1)+\ldots+C(X_m)$. $C(?)$ is defined to be ∞.

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

$$\min_{y \in \hat{V}_t^*} \{C(y) \mid X \Longrightarrow^* y\}$$

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: $C(y) = C(S(y))$.

---

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

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

$$\min_{y \in V_t^*} \{C(y) \mid A \Rightarrow^* ya\ldots\}$$

If $A \not\Rightarrow^* \ldots a \ldots$, then E(A,a) = ?

Algorithms which compute the S and E tables may be found in [6].

We are now ready to present the error-correction algorithm. It will compute a string "Insert" to be inserted to the immediate left of the symbol currently flagged as being in error. Let the parse stack be $X_n \ldots X_1$ (X_n is the stack top), and let the erroneous input symbol be 'a'. The parser will call the following algorithm:


```
function LL_Insert(Xn...X1,a) : Insert;
   { Xn...X1 is the LL(1) parse stack,
     a is the error symbol,
     Insert is the string to be inserted as a correction,
     Prefix is a least-cost prefix derivable from the stack
       symbols already processed                  }
Insert := ?; Prefix := λ;
for i := n downto 1 do
  if C(Prefix) ≥ C(Insert)
  then {No cheaper insertion is possible}
    return(Insert);
  if C(Prefix cat E(Xi,a)) < C(Insert)
  then {A cheaper insertion has been found}
    Insert := Prefix cat E(Xi,a);
  Prefix := Prefix cat S(Xi);
 end {for}
return(Insert)
end {LL_Corrector}
```

The function considers, in turn, individual parse stack symbols. When symbol X_i is considered, a possible correction is $S(X_n \ldots X_{i+1})$ *cat* $E(X_i,a)$. That is, $S(X_n \ldots X_{i+1})$ can be used to match stack symbols $X_n \ldots X_{i+1}$, then $E(X_i,a)$ can be used to allow 'a' to be matched. This string is adopted *if* it is cheaper than

(b) Assume LL_Insert extended as described in [6] is used with an LL(1) parser.
Then each invocation of the extended LL_Insert algorithm can be performed in constant bounded time and space.

□

Note that in case (b), the size of the Insert string returned by LL_Insert is not necessarily constant bounded in size. This does not contradict the results of the theorem however because the extended algorithm can specify the insertion by merely returning the index of the stack symbol which is to be used to generate the error symbol.

3. A Locally Least-Cost Correction Algorithm

We now add deletion operations to the correction process. Assume a user-supplied deletion cost function is available where for $a \in \hat{V}_t$, $D(a) \geq 0$ is the cost of deleting 'a'. $D(\lambda)$ is defined to be 0 and $D(\$)$ 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 further that 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 LL_Insert as a subroutine, computes locally least-cost corrections for LL(1) parsers.

```
function LL_Corrector(Xn...X1,b1...bm) : (Del,Insert);
{ Xn...X1 is the LL(1) 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(LL_Insert(Xn...X1,bI)) + D(b1...bI-1)
    < C(Insert) + D(b1...bDel)
  then begin {A better correction has been found }
    Insert := LL_Insert(Xn...X1,bI);
    Del := I - 1
  end
 end {For}
end {LL_Corrector}
```

the current correction string. Stack symbols are considered until the stack bottom is reached or until no cheaper insertion is possible (because Prefix, which must begin any new correction, is at least as costly as the current correction string). If none of the stack symbols can derive the error symbol, LL_Insert returns '?', the initial value of Insert.

Because the entire LL(1) parse stack is examined (if necessary), it is easy to see that LL_Insert will find a valid insertion if one exists. Further, the definitions of the S and E functions guarantee that the value of Insert returned is least-cost. Thus we have the following result from [6]:

Theorem 2.1

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

□

In practice, LL(1) 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[5]. For bounded depth LL(1) parsers it is easy to establish that each invocation of LL_Insert requires only a constant-bounded amount of time. In the general case, to parse an input x\$, a maximum stack depth of O(|x|) may be required. In this case a variant of LL_Insert can be employed. As detailed in [6], we can maintain an array of pointers to the top (i.e., uppermost) occurrence of each vocabulary symbol in the parse stack. Only those stack locations pointed to by the array need to be examined by LL_Insert. This follows from the observation that if the error symbol is to be derived by a symbol Y on the stack, the uppermost occurrence of Y can clearly be used. Because only a fixed number of stack locations (bounded by $|\hat{V}|$) need to be processed, LL_Insert can still execute in constant-bounded time. Thus the following result from [6] can be established:

Theorem 2.2

(a) Assume LL_Insert as defined above is used with a bounded depth LL(1) parser.
Then each invocation of LL_Insert requires constant bounded time and space.

---

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

LL_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 current 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 <u>any</u> correction is found, we tend to converge rapidly to the locally optimal correction. Indeed, as discussed in section 4, our tests indicate that the costs involved in computing locally optimal corrections are quite reasonable and apparently no real problem in actual production compilers.

In cases where we wish to perform <u>error-recovery</u> rather than error-correction, the queuing of input symbols required by LL_Corrector may be an undesirable complication. Besides actually maintaining the queue (which is itself a space and time overhead), we must also worry about formatting a source listing with appropriate error diagnostics. This usually implies that the source images associated with queued symbols must also be saved so that suitable messages may be generated once a correction is determined.

For error correctors this extra complexity must be borne in the interests of obtaining the best available repairs. Since error recovery routines are primarily interested in simply restarting parsing after a syntax error, the incremental advantage in using a locally least cost correction to restart an LL(1) parser may not be worth the costs involved. In such cases a variation of LL_Corrector can be used instead. As usual, the algorithm will examine and delete input symbols until a finite-cost correction is found. Thereafter, it will examine subsequent input symbols <u>only</u> if deleting them leads to progressively better (i.e., cheaper) corrections. Once an input symbol is reached whose deletion does not lead to a better correction, the modified LL_Corrector returns.

The chief advantage of such an approach is that input symbols need never be queued since we never look beyond the first non-deleted input symbol. Of course, this modification does not always find a locally optimal correction but it always does at least as well (and often better than) the original insertion-only corrector with almost no additional complexity or computational overhead. Thus for recovery purposes it represents an especially nice balance between simplicity of construction and quality of performance. As such, it appears to compare favorably with other LL(1) recovery techniques ([10], [11]).

The following establish the correctness, local optimality and robustness of the LL_Corrector routine.

Theorem 3.1

Assume that some LL(1) parser for a grammar, 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 LL_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 LL_Insert routine.

□

Corollary 3.2

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

Proof:

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

□

We now turn our attention to efficiency issues. In considering the space and time requirements of LL_Corrector, it is important to note that the corrector and associated parser will almost certainly not be used in their full generality. As noted earlier, a bounded depth parse stack will almost certainly be used by the LL(1) parser. So too, deletion costs of zero, although allowed by our model, seem never to be used (since they make wholesale deletions far too easy). It is easy to establish that LL_Corrector, when used with a bounded depth parse stack and strictly positive deletion costs, is linear in operation.

Theorem 3.3

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

Proof:

Each invocation of LL_Insert requires constant time (by Theorem 2.2). The size of Insert returned by LL_Insert can also be constant bounded (because each stack state contributes a piece of bounded size). Consider each iteration of LL_Corrector as it processes input symbols. Until LL_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 to be deleted and each such symbol is charged only once. Once LL_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.

□

Recall from section 2 that non-bounded depth LL(1) parse stacks can be accommodated by extending the LL_Insert routine. Similarly, deletion costs of zero can be handled by preprocessing the input (when the first syntax error is discovered) so that when LL_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 LL_Insert($X_n \ldots X_1$,b), we need only delete up to the <u>first</u> <u>occurrence</u> of b in the remaining input (to which we have a pointer). This means LL_Corrector needs to only invoke LL_Insert at most $|V_t|$ times per error and since at most $O(|x|)$ errors are possible, the following can be established.

<u>Theorem 3.4</u>

Assume an LL(1) parser uses LL_Corrector extended as outlined above.

Then any input string x$ can be parsed and (if necessary) corrected in $O(|x|)$ time and space.

<u>Proof</u>: Follows from Theorems 2.2 and 3.3 and the above discussion.

□

Because of the extra overhead and complexity the above extension entails, it would certainly have a <u>worse</u> average-case behavior than the original LL_Corrector. Thus we do not expect that it would ever be used in practice. Nevertheless, it is of value in showing that efficient (i.e., linear) LL(1) correctors can always be constructed.

4. Implementation and Test Results

The LL_corrector algorithm has been implemented and tested on a number of LL(1) grammars, including ones for Pascal and a variant of ALGOL 60. The speed of table generation was quite acceptable, requiring about 60 seconds to compute and store the S and E tables for the ALGOL grammar ($|V_t|=65$, $|V_n|=91$, $|P|=174$) and about 70 seconds for the Pascal grammar ($|V_t|=67$, $|V_n|=133$, $|P|=251$) on a Digital Equipment VAX-11/780. Total sizes for the D, S and E tables were 37K bytes for the ALGOL grammar and 40K bytes for the Pascal grammar. Only a small fraction of the tables need be kept in main storage. The D and S tables, which are fairly small, would normally be stored in main memory, but the E table, which accounts for most of the total space requirement, can easily be kept in secondary storage, since only one column of the table is needed for each call to LL_Insert. Execution of the correction algorithm is very fast, requiring an average of 9 milliseconds per correction (excluding file access time).

As mentioned in section 3, usually very few iterations of the loop in LL_Corrector (i.e., calls to LL_Insert) are needed to determine a locally optimal correction. Our measurements indicate that with fairly well-tuned correction costs, a deletion is considered in only about 50% of the corrections. In very few

cases is deletion of more than one symbol considered. Thus deletions have only a small impact on the speed of the corrector.

The following short program (adapted from [9]) provides examples of the kinds of corrections effected by LL_Corrector. The correction costs used are listed in appendix A.1. 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.   1: i + j > k + l + 4
          ↑
7.   then write ( i ; )
     ↑              ↑ ↑
8.   b 1, 2] := 3 * ( i/ + j ;
       ↑↑ ↑ ↑            ↑   ↑
9.   if i = 1 then then goto 1;
                   ↑
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 + 1 + 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, '*if*' should probably be inserted before 'i'. Such a correction *cannot* be performed by LL_Corrector (or most other correction techniques) because 'i' has already been consumed by the parser when the error is detected. Some correctors ([8], [9]) advocate a "backward move" in such a situation but this can be very difficult in a one-pass compiler since symbols accepted by the parser may already have been translated. Graham, Haley and Joy [8] suggest that backward moves be limited to terminal symbols which have not yet been reduced (to guarantee that no semantic actions need to be "undone"). Unfortunately, this approach (designed for LALR(1) parsers) is not suitable for LL(1) parsers, as semantic actions,

initiated by *action symbols*, can occur at any point during a parse.

An interesting alternative is the use of *error productions* as discussed in [5]. The idea here is to augment a grammar with productions which *anticipate* certain syntax errors. Thus an expression might be allowed to begin a statement to provide for a missing statement header (e.g., an *if*, *while*, *case*, etc.). This approach has been tested with LALR(1) grammars and appears to be very effective. Because LL(1) grammars are more restrictive than LALR(1) grammars, error productions may be more difficult to use with LL(1) parsers. Nevertheless, such productions are an extremely promising way of dealing with problems such as missing statement headers because they can be employed without *any* changes at all to the parser and error corrector being used.

Another difficulty appears in line 8 in which ...b[1,2]... is probably intended. The difficulty here is that LL_Corrector seeks only *local* 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 "forward move" phase as suggested by ([8], [9], [12])). However once again this is a 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 LL_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 [2]. This approach, although as yet untested, seems to have great potential for improving the overall quality of the correction process. Other correctors ([3], [8]) utilize context-sensitive information by calling semantic routines to determine the appropriateness of a proposed correction. This method can be fairly effective but care is required to ensure that semantic routines called from an error corrector have no side effects (since proposed corrections are only tentative). A more serious difficulty is that knowledge of context-sensitive issues must be built directly into an error corrector (so that, e.g., it knows which semantic routines to call). Further, this must usually be done in an *ad-hoc* manner.

In this case, error productions are again an interesting alternative because they allow an extant corrector (such as LL_Corrector) to be employed without modification. The idea here is to add new symbols and productions to represent *some*

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 can be modified so that a '[' can follow an <array id> but *not* 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 grammar, although fairly straightforward, are extremely useful in enhancing the performance of LL_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 correction 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.1 and

A.2; another set may be found in [15]. The interested reader may find a more detailed discussion of cost selection in [14].

To evaluate the performance of our error-corrector, we adopted the criteria of Pennello and DeRemer [12]: 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, LL_Corrector, in the above example, performed 8 excellent corrections, 1 good correction and 2 poor corrections[6].

We compared LL_Corrector with the Simple Precedence corrector of Graham and Rhodes [9], the SLR(1) corrector of Tai [15], and the "insertion-only" LL(1) corrector of [6]. All four techniques were applied to a 63 statement ALGOL program from [14]. The correction costs used by LL_Corrector are listed in appendix A.2.

---

[6]Spurious errors, induced by an error corrector, are not included in error counts as they are not considered "real" user errors.

	Excellent	Good	Poor
LL(1) [6]	45%	26%	29%
SP [9]	40%	42%	18%
SLR(1) [15]	41%	51%	8%
LL_Corrector	61%	25%	14%

The performance of LL_Corrector is rather impressive and is certainly comparable, or superior to, the other correction algorithms. It is important to note that the performance of LL_Corrector on the above test program is exactly the same as that of LR_Corrector [4], an LR-based error corrector implementing the same locally least-cost model of correction. This emphasizes the value of having a high-level correction model in which details of the context-free grammar and parsing technique being used can be completely ignored.

More interestingly, the simplified version of LL_Corrector suggested in section 3 (which considers deletions only as long as progressively cheaper corrections are found), also performed exactly the same as LL_Corrector on the test program. In fact, in almost all of our tests, the simplifed LL_Corrector routine produced results identical to LL_Corrector. Only when presented with very ill-formed inputs (e.g., several extra right parentheses) did the modified LL_Corrector produce a non-locally optimal repair. This then suggests that this simplified routine can indeed be used as the basis of a very efficient and effective error recovery scheme.

In judging the above performance figures, it is important to note that the performance criteria used are rather subjective and open to a wide degree of interpretation. Thus, we adjudged a correction poor <u>whenever</u> it led to subsequent "spurious" errors. In cases where a "cluster" of errors appear, however, it is natural for LL_Corrector to sometimes do a correction <u>incrementally</u>, 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." LL_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 <u>poor</u> 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 LL_Corrector on the ALGOL test program is now: 61% excellent, 33% good and only 6% poor. These figures seem representative of LL_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 LL_Corrector's performance with that of the LL_Insert routine when it alone is used as an "insertion-only" error corrector. The difference in performance between the two is almost wholly[7] attributable to the fact that deletion operations are eschewed by LL_Insert. As indicated above, the main difference between the two is an increase of about 15% in the number of "poor" corrections attributed to LL_Insert. 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 [6] that an

---

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

insertion-only corrector can be used in practice with satisfactory results.

5. 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 language being processed. The corrector is usable with any LL(1) grammar and is automatically generable. The technique can be guaranteed to handle correctly any input and all corrections are locally optimal. In cases of practical interest linearity can easily be established. The correction algorithm can be simplified by eliminating the queuing of input symbols as deletions are considered. The resulting routine performs almost as well as the original and seems especially well suited for use in error recovery.

Test results are equally encouraging. The corrector has little impact on parsing speed even when processing very ill-formed inputs. Primary memory requirements are minor 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 [5], judiciously chosen error productions seem to be of great value in handling certain difficult cases. Ways of increasing the context available in choosing corrections (as suggested, e.g., in [13]) without unduly impacting the structure or efficiency of the host compiler are of interest. Because of the predictive nature of LL(1) parsing, this should be easier (and cheaper) to do than in LR-based error correctors. Also, methods which include context-sensitive considerations (e.g., type and scoping rules) in the correction process, as described in [2], have the potential to greatly enhance overall correction quality and certainly deserve careful study.

In summary, the LL(1) error corrector presented occupies a middle position in the spectrum of known error correctors. 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 repertoire of context-free correction techniques and a valuable tool in building modern compilers.

Acknowledgments

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

Appendix A.1 - 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

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

2. B.A. Dion, "Locally least-cost error correctors for context-free and context-sensitive parsers," Comp. Sci. Dept., Univ. of Wisconsin-Madison, Ph.D. thesis, Tech. Rep. 344, Dec. 1978.

3. S. Feyock and P. Lazarus, "Syntax-directed correction of syntax errors," Software Practice and Experience Vol. 6, pp. 207-219, 1976.

4. C.N. Fischer, B.A. Dion and J. Mauney, "A locally least-cost LR-error corrector," Comp. Sci. Dept., Univ. of Wisconsin-Madison, Tech. Rep. 363, submitted to ACM Trans. Prog. Lang. and Sys., 1979.

5. C.N. Fischer and J. Mauney, "On the role of error productions in syntactic error correction," Comp. Sci. Dept., Univ. of Wisconsin-Madison, Tech. Rep. 364, submitted to Computer Languages, 1979.

6. C.N. Fischer, D.R. Milton and S.B. Quiring, "Efficient LL(1) error correction and recovery using only insertions," to appear in Acta Informatica, 1979.

7. C.N. Fischer, K.C. Tai and D.R. Milton, "Immediate error detection in strong LL(1) parsers," Inform. Proc. Letters vol. 8, no. 5, pp. 261-266, 1979.

8. S.L. Graham, C.B. Haley and W.N. Joy, "Practical LR error recovery," Proc. of the Sigplan Sym. on Compiler Construction, in Sigplan Notices, vol. 14, no. 8, pp. 168-175, 1979.

9. S.L. Graham and S.P. Rhodes, "Practical syntactic error recovery," Commun. ACM vol. 18, pp. 639-650, 1975.

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

11. A.B. Pai and R.B. Kieburtz, "Global context recovery: a new strategy for parser recovery from syntax errors," Proc. of the Sigplan Sym. on Compiler Construction, in Sigplan Notices, vol. 14, no. 8, pp. 158-167, 1979.

12. T.J. Pennello and F.L. DeRemer, "A forward move algorithm for LR error recovery," in Conf. Rec. 5th Annual ACM Sym. Principles of Programming Languages, 1978, pp. 241-254.

13. D.A. Poplawski, "Error recovery for extended LL-Regular parsers," Comput. Sci. Dept., Purdue Univ., Ph.D. thesis, Aug. 1978.

14. S.P. Rhodes, "Practical syntactic error recovery for programming languages," Dept. Comput. Sci., Univ. of California, Berkeley, Ph.D. thesis, Tech. Rep. 15, 1973.

15. K.C. Tai, "Syntactic error correction in programming languages," IEEE Trans. Software Eng. vol. SE-4, no. 5, pp. 414-425, 1978.