A Forward Move Algorithm for LL and LR Parsers

Jon Mauney
Charles N. Fischer

University of Wisconsin-Madison

1. Introduction

A wide variety of algorithms have been suggested for the repair of syntactic errors in a computer program. Since there is usually more than one possible repair for any syntax error, many algorithms employ a cost function to guide the the repair, and some [1,3,4,6], guarantee that the repair chosen will be least-cost, according to some definition. (The others, although guided by costs, do not guarantee least-cost in all cases.) Fischer et al. [4,6,7] define a "locally least-cost" repair using insertions and deletions, and provide algorithms for LL and LR parsers. A locally least-cost repair is a least-cost sequence of deletions and insertions such that one more symbol in the original string will be accepted by the parser. Backhouse [2,3] uses a similar definition. In both cases, the repair algorithms operate by examining a single symbol in the input at any time.

There are situations in which a repair algorithm needs more information than is provided by a single symbol. For example, in the Pascal program fragment:

```
... ; a := b c ...
```

the parser would announce error upon reading 'c', and the possible repairs would include:

```
... ; a := b + c ...
... ; a := b ; c ...
... ; a := b [ c ...
```

One of these repairs will have the lowest cost, and will always be chosen by the repair algorithm. (If several repairs have equally low cost, the algorithm must pick one. There is no advantage in trying to choose non-deterministically.) However, if the algorithm looks ahead at additional input symbols, it may gather information which distinguishes the current situation from the other possibilities:

```
... ; a := b  c ;  ...
... ; a := b  c := ...
... ; a := b  c ]  ...
```

The algorithm can then choose the best repair in each case.

Research supported in part by NSF grant MCS78-02570

The algorithms presented by Graham and Rhodes [10], Pennello and DeRemer [14], Pai and Kieburtz [13], and Graham, Joy, and Haley [8] use various amounts of additional lookahead, but do not guarantee a least-cost repair in all cases. The action of the algorithms may also be limited by the presence of a second error in the input. The ultimate in least-cost repair models is the globally minimum-distance model of Aho and Peterson [1]. Such a model guarantees the minimum number of changes to the entire sentence. Unfortunately, the best algorithm known is cubic in the length of the sentence, and is therefore considered impractical.

We suggest a middle ground between locally and globally least-cost models; the repair algorithm will select some region of the sentence and find the least-cost repair to that region which will allow the parse to continue through the region. Levy [11] and Tai [16] have described such a repair model as "locally least-cost", in contrast to global least-cost models. Since we have previously considered locally least-cost repair to involve only a single symbol at a time [4,6,7], we will use the phrase "regionally least-cost" to describe the area between single-symbol and global models.

In the examples above, a regionally least-cost repair algorithm with a region size of two symbols would choose the repairs suggested (assuming a reasonable set of repair costs). For a more involved example, let us look at another portion of a Pascal program:

```
if i<j
  A[i  := j
else A[j] := i;
```

The parser will announce error upon encountering the first 'A', and the possible repairs include inserting 'then' and inserting an arithmetic operator. Again, a well-tuned locally least-cost repair algorithm would choose the most likely repair, and would occasionally be wrong. A regionally least-cost algorithm with a region of three symbols would see "A[i", and would really be no better off than the locally least-cost algorithm. If the region is extended to four symbols, the algorithm must cope with a second error. The options now include inserting "then" before the "A" and "]" before the ":=", inserting an operator before "A" and replacing the "[" with "then", and inserting an operator before "A" and replacing the ":=" with an operator. Extending the region further, to five or six symbols, shows that replacing the ":=" is undesirable, as it leads to still more errors. The choice, then, is a matter of where to insert the "then", and this will be resolved by the costs of the other repairs (inserting "]" versus inserting an operator and replacing "[").

The previous example illustrates that repair algorithms which employ a forward move must deal with clusters of errors, and that the size of the region greatly affects the repair chosen. The definition of a regionally least-cost repair implies repair of any errors within the region. Tai's MCL(k) model [16] is an example of a regionally least-cost repair model, with a particular method of determining the region size. We present an algorithm which finds a regionally least-cost repair for any region size, and discuss ways of determining region size, as well as results of an initial implementation of the algorithm. The algorithm is adaptable for use with LL(1) and LR(1) parsers.

2. Regionally Least-Cost Repair

We begin by defining regionally least-cost repair. As usual, we will assume a context-free grammar, $G=(V_t,V_n,S,P)$; L(G) is the language generated by G, and Pr(G) is the set of prefixes of sentences in L(G). The problem is to repair an arbitrary string, $x \in V_t^*$, into a string, $y \in V_t^*$, that is a sentence in L(G). We will use the three primitive edit operations insert, delete, and replace, and we will search for a repair of least cost, based on cost vectors for the three operations:

IC(a) gives the cost of inserting terminal symbol a
DC(a) gives the cost of deleting terminal symbol a
RC(a,b) gives the cost of replacing a with b

We require that all costs be non-negative. We will also assume that at most one replacement or deletion is made at each input position. This assumption is equivalent to requiring that the costs satisfy "triangle inequalities":

For all $a,b,c \in V_t$

$RC(a,b)+RC(b,c) \geq RC(a,c)$
$RC(a,b)+DC(b) \geq DC(a)$
$IC(a)+RC(a,b) \geq IC(b)$

It is convenient to assume that $RC(a,a)=\emptyset$.

Based on the cost functions, we define two other functions that extend costs to nonterminals, strings and derivations:

$C(\lambda) = \emptyset$; (λ denotes the empty string.)

$C(a_1 \ldots a_n) = IC(a_1) + \ldots + IC(a_n)$, for $a_i \in V_t$

$C(A) = \min \{ C(x) \mid x \in V_t^*, A \Rightarrow^* x \}$

$C(X_1 \ldots X_n) = C(X_1) + \ldots + C(X_n)$, for $X_i \in V$

For $A \in V_n$, $a \in V_t$, $Derive(A,a) = \min[\{ \infty \} \cup \{C(xy)+RC(a,b) \mid A \Rightarrow^* xby \}]$ $x,y \in V_t^*$, $b \in V_t$

For $A,B \in V_n$, $Derive(A,B) = \min[\{ \infty \} \cup \{C(xy) \mid A \Rightarrow^* xBy \}]$ $x,y \in V_t^*$

We are now ready to define a regionally least cost repair.

Definition: A modification, M, is a series of edit operations, $E_1E_2\ldots E_n$, where each E_i is a series of insertions, and an optional delete or replace operation. The string resulting from the application of the modification, M, to a string x, $|x|=n$, is written M(x). The cost of the modification, C(M(x)), is the sum of the costs of the edit operations.

Definition: Given two strings x, $y \in V_t^*$, with x in Pr(G), a repair of y following x, is a modification, M, such that xM(y) is in Pr(G).

Definition: A regionally least-cost repair of y following x is a repair, M, of y following x such that for any other such repair, N, of y following x, $C(N(x)) \geq C(M(x))$.

3. An algorithm for globally least-cost repair

We will develop the algorithm first as a globally least-cost repair algorithm. Then in the next section, we will restrict it to the more feasible regionally least-cost case.

Aho and Peterson perform their "least errors parse" by adding to the grammar error productions, which simulate modifications to the input. We will take a complementary approach, and use a modified parser on the original grammar. The modified parser will be able to: advance the parser state as if additional symbols were in the

input (simulate insertions), consume input without changing the parse state (deletions), and advance the parser state as if one symbol were in the input, while consuming some other symbol (replacements). These changes to the parser are equivalent to the error productions used by Aho and Peterson, and render the grammar highly ambiguous; a general context-free parser is therefore necessary.

For our parser, we choose the algorithm of Graham, Harrison and Ruzzo [9]. This parsing algorithm produces a triangular matrix, each cell of which contains a set of "dotted productions", $A \rightarrow \alpha \cdot \beta$, representing the possible parses of a corresponding substring of the input. The dot indicates that part of the production, α, has been used to match input symbols. The position of a cell in the matrix indicates the portion of the input covered; cell i,j covers symbols i+1 through j, inclusive. Cells that match a longer substring of the input are created by "pasting together" two existing cells; elements in cell i,j are found by pasting cell i,k to cell k,j for $i<k<j$. The parse is also advanced by pasting cells to the input symbols.

We introduce error repair by extending these pasting operations (and the "Predict" function), and attaching a running cost to each dotted production. When the parse is completed, we extract the repair from the matrix in much the same way as a parse is extracted in ordinary use. The remainder of the algorithm is unchanged.

Informally, the Graham-Harrison-Ruzzo algorithm will move the dot if the symbol immediately to its right is matched. For example, if we have the dotted production $A \rightarrow \alpha \cdot B\beta$, and some production $B \rightarrow \gamma$ has been satisfied, then we can move the dot, yielding $A \rightarrow \alpha B \cdot \beta$. In order to accommodate λ-productions, the dot is also moved across symbols which can derive λ. The result is a set of dotted productions, each with the dot in a different location. The original pasting operators are:

For Q a set of dotted productions and R a set of symbols:

$$Q^{x}R = \{A \rightarrow \alpha B\beta \cdot \gamma \mid A \rightarrow \alpha \cdot B\beta\gamma \in Q, \beta \stackrel{*}{\Rightarrow} \lambda, \text{ and } B \in R\}$$

$$Q^{*}R = \{A \rightarrow \alpha B\beta \cdot \gamma \mid A \rightarrow \alpha \cdot B\beta\gamma \in Q, \beta \stackrel{*}{\Rightarrow} \lambda, \text{ and } B \stackrel{*}{\Rightarrow} C \text{ for some } C \in R\}$$

For Q and R sets of dotted productions:

$$Q^{x}R = \{A \rightarrow \alpha B\beta \cdot \gamma \mid A \rightarrow \alpha \cdot B\beta\gamma \in Q, \beta \stackrel{*}{\Rightarrow} \lambda, \text{ and } B \rightarrow \delta \cdot \in R\}$$

$$Q^{*}R = \{A \rightarrow \alpha B\beta \cdot \gamma \mid A \rightarrow \alpha \cdot B\beta\gamma \in Q, \beta \stackrel{*}{\Rightarrow} \lambda, B \stackrel{*}{\Rightarrow} C \text{ for some } C \in V_t^{*} \text{ and } C \rightarrow \delta \cdot \in R\}$$

The x and * products differ in that x considers only direct derivations, while * considers indirect derivations.

We can simulate insertions into the program by moving the dot across any symbol, X, as if it derived λ, at a cost of C(X). This change to the parsing algorithm is equivalent to adding error productions to the grammar which allow all symbols to derive λ. Those symbols that derive λ in the original grammar will have $C(X)=\emptyset$; in such cases, the action of the parser will be the same as it would in the original algorithm. To simulate a deletion, we paste a dotted production to an input symbol, a, without moving the dot, at a cost of DC(a). Thus a symbol is consumed without advancing the parse state. We can now present our modified pasting operators.

If Q is a set of dotted productions and R is a set of terminal symbols, then define

$$Q^{x}R = \{ A \rightarrow \alpha B\beta \cdot \gamma ; c \mid A \rightarrow \alpha \cdot B\beta\gamma ; c' \in Q, B \in R, c = c' + C(\beta) \} \cup \{ A \rightarrow \alpha \cdot \beta ; c \mid A \rightarrow \alpha \cdot \beta ; c' \in Q, c = c' + DC(B), B \in R \}$$

$$Q*R = \{\ A \rightarrow \alpha B\beta\cdot\gamma;c \mid A \rightarrow \alpha\cdot B\beta\gamma;c' \in Q,\ Derive(B,D) \neq \infty,\ D \in R,\ c = c'+C(\beta)+Derive(B,D)\ \}$$
$$\cup\ \{\ A \rightarrow \alpha\cdot\beta;c \mid A \rightarrow \alpha\cdot\beta;c' \in Q,\ c = c'+DC(B),\ B \in R\ \}$$

If Q and R are sets of dotted productions, then define

$$Q \times R = \{\ A \rightarrow \alpha B\beta\cdot\gamma;c \mid A \rightarrow \alpha\cdot B\beta\gamma;c_1 \in Q,\ B \rightarrow \delta\cdot;c_2 \in R,\ c = c_1 + c_2 + C(\beta)\ \}.$$

$$Q*R = (\ A \rightarrow \alpha B\beta\cdot\gamma;c \mid A \rightarrow \alpha\cdot B\beta\gamma;c_1 \in Q,\ Derive(B,D) \neq \infty,\ D \rightarrow \delta\cdot;c_2 \in R,\ c = c_1+c_2+Derive(B,D)+C(\beta)\ \}.$$

We must also modify the Predict function, which, in the original algorithm, sets up chain rules and insures that the contents of subsequent cells will legally follow the parse so far. In our algorithm, Predict sets up insertions as well. For R a subset of V_n

$$PREDICT(R) = \{\ C \rightarrow \alpha\cdot\beta;c \mid C \rightarrow \alpha\beta \in P,\ B \stackrel{*}{\Rightarrow} \gamma C\delta,\ B \in R,\ c = C(\alpha)\ \}$$

For R a set of dotted productions,

$$PREDICT(R) = PREDICT(\ \{B \mid A \rightarrow \alpha\cdot B\beta \in R\}\ ).$$

The effect of these extensions to the parsing algorithm is the same as the effect of adding Aho and Peterson's error productions; the sets in the parse matrix are isomorphic to those that would be obtained using the modified grammar. Therefore, the correctness and complexity of the parser should be unchanged by the modifications. The proof of the correctness of the algorithm [12] follows the same lines as the proof in [9], with a different definition of what it means to match a string. The only danger is the additional cost component of the dotted productions: since the cost is potentially unbounded, the presence of dotted productions that differ only in the cost could cause the size of a cell to be unbounded. However, it is easy to show that if two dotted productions in a set differ only in the cost component, then the one with higher cost can never participate in a least cost repair; any parse can be made cheaper by using the other dotted production. Therefore, a higher cost duplicate can always be discarded, and the size of the cell is not affected by the presence of cost components.

4. An algorithm for regionally least-cost repair

We have presented an algorithm that finds a least-cost repair of an entire program, in time proportional to the cube of the length of the program. We do not propose that it be used as such. Instead, we intend that the repair algorithm be called only when needed, and then only to repair a reasonably sized region of the program. A linear-time parser, such as LL(1) or LR(1), will be used for the major portion of the program (for the entire program if there are no errors).

Instead of producing a sentence of the language, we want our algorithm to produce a string which can follow the input already accepted by the parser; therefore we will use a grammar that describes the legal continuations. Such a grammar can be derived from the state of the parser. If an LL(1) parser is used, this task is easy: the parse stack describes the expected suffix, and we replace the starting production with the production S --> stack. For an LR(1) (or SLR or LALR) parser, we can use the technique described in [4] to derive a regular expression that describes the legal suffixes. From this regular expression we

can easily derive an equivalent context free grammar, which will be added to the original grammar for the purpose of repair.

Once the parse/repair algorithm has started, it can stop at any point; after each iteration of the main loop, the region of least-cost repair has been extended over one more input symbol. Thus, the algorithm can be used to find a least-cost repair over a fixed-sized region, or the size of the region can be dynamically controlled. A repair over a region of fixed size has an advantage in that it requires a fixed amount of time to compute, but the fixed size may be too small for some error situations and unnecessarily large for others. We are then faced with the question of how to choose the region size. Levy [11] suggests that the region should extend until all of the plausible repairs are "equivalent" in that all suffixes of one repaired string are suffixes of the other repaired strings. This criterion is appealing, because it implies that no matter which repair is chosen, there can be no effect on the legality of the remaining input. Unfortunately, in most programming languages, the test is impractical. For example, in Pascal one of the repairs may include insertion of begin, and legal suffixes will thus contain an extra end, which other repairs will not accept. Until the end of the input is reached, the repairs cannot be equivalent. Tai [16] suggests extending the region over clusters of errors, until a sequence of k correct symbols are seen. This criterion avoids continuing to the end of the input -- unless the cluster of errors extends that far -- but the region beyond the last error is always fixed, and may be too large or small in some cases. We are investigating a number of criteria for dynamic region sizes, including Tai's MCL(k) and a weakened version of Levy's equivalence.

In order to obtain better repairs by using a dynamic region, we run the risk of extending the region to the entire program; in that case our algorithm is cubic in the size of the program. In practice this may not be a problem. In fact, if the expected size of a region is constant, the expected time to compute a repair may also be constant, if the distribution is reasonable. For instance, if the size of a region is some fixed minimum, k, plus a variable part that follows an exponential distribution, $P(m)=Ce^{-Dm}$, then the expected time to repair that region is less than $C1+C2/(1-e^{-Dn-D})$, which is bounded by a constant as the size of the program, n, grows. Thus, in the average case the total time to parse and repair a program is linear in the length of the program. As we experiment with dynamic regions, we will measure the distribution of region sizes.

After the repair has been chosen, control is returned to the linear-time parser. Repairs to the program can be effected in two ways. The repaired string can be physically placed into the input buffer and reparsed, or the state of the parser can be reset, using information from the parse matrix.

As an example, suppose we have the following grammar:

```
S --> E Etail
E --> a | ( E )
Etail --> + E Etail | λ
```

Assume all terminals have insertion cost = 1 and deletion cost = 2. If we try to parse the input "a + a a)" using a strong LL(1) parser, an error will be detected at the third 'a', with the parse stack containing 'Etail'. The repair algorithm will proceed as follows:

1. 'a' can be matched by the 'E' in Etail --> + E Etail at cost C('+')=1. 'a' can also be matched directly by E --> a, and from E --> (E), at a cost of C('(')=1. Finally, 'a' can be deleted, at cost DC('a')=2. Cell Ø,1 of the parse matrix will contain these dotted productions:

```
Etail --> + E · Etail ; 1
E     --> a · ; Ø
E     --> ( E · ) ; 1
S     --> · Etail ; 2
      {deletion of a}
```

2. ')' can be pasted to a previous match, such as E --> (E·). It can also be deleted. Cell Ø,2 of the parse matrix will contain

```
E     --> ( E ) · ; 1
  {paste to result of matching 'a'}
S     --> Etail · ; 5
  {paste to result of deleting 'a'}
Etail --> + E · Etail ; 3
  {deletion of ')'}
E     --> a · ; 2
  {ditto}
E     --> ( E · ) ; 3
  {ditto}
S     --> · Etail ; 4
  {deletion of 'a' and ')'}
Etail --> + E · Etail ; 2
  {paste E-->(E)·;1 to predicted
   Etail-->+·E Etail;1}
S     --> Etail · ; 2
  {paste E-->(E)· ;1 to
   S-->·Etail;Ø, Derive(Etail,E)=1}
```

For simplicity, we have not shown all of the elements of the cells, nor all the cells in the matrix. The elements shown for cell Ø,2 illustrate that the same dotted production may be entered twice, with different costs; a lower cost version will replace a higher cost. Although the lowest cost element in cell Ø,2 is "E-->(E)·;1", that cost only includes the repairs necessary to match the input -- insert '(' before 'a) -- and not the repairs necessary to follow the previously accepted input -- insert '+'. To insure that all repairs are included, we must start from an element involving the goal production -- S --> Etail · ;2 in this case -- or one which is directly predicted by the goal production -- Etail --> + E · Etail ;2. A traceback from one of these dotted productions will show that the regionally least-cost repair is to insert '+(' before the 'a'. A locally least-cost algorithm would have handled the error in two steps: first inserting '+' before the 'a', then deleting the ')' when the parser again announces error. This combined repair is inferior to the regionally least-cost repair.

5. Implementation results

In the introduction we presented a series of situations which show that a locally least-cost algorithm must occasionally choose a poor repair. Let us examine a few more cases in which we have found the regionally least-cost algorithm to perform better than the locally least-cost. These examples are adapted from Ripley's and Druseikis's collection of Pascal errors [15].

```
program p (input, output);
funtion f (var x: integer): boolean;
     begin
     end;
begin
end.
```

In this example, an error is announced when 'funtion' is read. Without a spelling corrector, replacing 'funtion' with 'function' is not plausible, so we must make do with insertions and deletions. A likely locally least-cost repair is insertion of 'const' or 'type', which will cause a cas-

cade of spurious errors. Again, a regionally least-cost will see the additional repairs necessary if 'const' is inserted, and will instead insert 'function' and delete either 'function' or 'f'. This example also emphasizes the effect of region size; a region of size less than nine will not extend as far as the return type, and the algorithm will have no information with which to distinguish insertion of 'procedure' from 'function'.

```
program p (input, output);
var a, b, : real; i, j : integer;
begin end.
```

The error is an extra comma, but the parser will not detect error until the colon is read. Without a backward move, there is no way to remove the comma, so we must either insert another identifier, or delete the colon. Deleting the colon is likely to be chosen by a locally least-cost algorithm, causing additional errors. Again, the regionally least-cost algorithm will choose the preferred repair: in this case, inserting an identifier.

In the two examples above, the repairs suggested as likely for a locally least-cost algorithm were, in fact, chosen by the algorithm described in [4], using a set of well-tuned costs. On the error programs provided by Ripley and Druseikis, we found that about 28% were repaired poorly by the locally least-cost algorithm. Of these, over 80% were improved upon by the regionally least-cost algorithm. Of course, not all of the repairs were as good as in the examples above. The sample programs include a number of systematic misuses of the language, such as declaration sections in the wrong order (which are best handled by error productions [5,8]), and serious lexical errors, such as improper comment delimiters. In such cases, a regionally least-cost algorithm makes less of a mess than a locally least-cost algorithm.

The repair algorithm requires careful implementation if reasonable speed is expected. The effect of region size on computation time is overshadowed by the effect of grammar size, for moderate regions. Our prototype, a straightforward implementation, computes a five symbol repair for Pascal in approximately five seconds (on a VAX-11/780). We are confident that a more careful implementation could be made significantly faster. Even so, the algorithm will be slower than locally least-cost techniques. However, further speed-up is available if the regionally least-cost algorithm is not called for every error. Since the locally least-cost algorithm does well in most cases, we can try out the local repair first. Parsing ahead several symbols to validate the repair is quick (in a one-pass compiler, semantic actions will be disabled), and if additional errors are found, the regionally least-cost algorithm is invoked. If no additional errors are found, the local repair stands, and parsing/compilation resumes in earnest. Since the greater computational cost of a regionally least-cost repair is incurred only in those cases in which it is needed, the average speed is improved with no loss of repair quality.

6. Conclusion

We have presented a model of error-repair using a forward move. This model provides a formal, language-level definition of how repairs are chosen, using the idea of regional least-cost. The algorithm to compute such corrections is linear in the size of the grammar, and cubic in the size of the region. Even if region size is variable, in the average case the total complexity of the parse/repair package is essentially linear.

7. References

[1] Aho, Alfred V. and Thomas G. Peterson , "A minimum distance error correcting parser for context-free languages," SIAM Journal of Computing 1, 4, pp. 305-312 (1972).

[2] Anderson, S. O. and Roland C. Backhouse, "Locally least-cost error recovery in Earley's algorithm," ACM Transactions on Programming Languages and Systems 3, 3, pp. 318-347 (July 1981).

[3] Backhouse, Roland C., Syntax of Programming Languages, Theory and Practice, Prentice-Hall (1979).

[4] Fischer, Charles N., Bernard A. Dion, and Jon Mauney, "A Locally Least-Cost LR Error-Corrector," ACM Transaction on Programming Languages and Systems, (to appear).

[5] Fischer, Charles N. and Jon Mauney, "On the role of error productions in syntactic error correction," Computer Languages 5, pp. 131-139 (1981).

[6] Fischer, Charles N., Donn R. Milton, and Jon Mauney, "A locally least-cost LL(1) error corrector," Tech. Report #371, University of Wisconsin (August 1979).

[7] Fischer, Charles N., Donn R. Milton, and Sam B. Quiring, "Efficient LL(1) error correction and recovery using only insertions," Acta Informatica 13, 2, pp. 141-154 (1980).

[8] Graham, Susan L., Charles B. Haley, and William N. Joy, "Practical LR error recovery," Sigplan Notices 14, 8, pp. 168-175 (1979)

[9] Graham, Susan L., Michael A. Harrison, and Walter L. Ruzzo, "An Improved Context-Free Recognizer," ACM Transactions on Programming Languages and Systems 2, 3, pp. 415-462 (July 1980).

[10] Graham, Susan L. and Steven P. Rhodes, "Practical syntactic error recovery," Communications of the ACM 18, pp. 639-650 (1975).

[11] Levy, J. P., "Automatic correction of syntax errors in programming languages," Acta Informatica 4, pp. 271-292 (1975).

[12] Mauney, Jon, "Least-cost error repair using extended right context", Ph.D. thesis, in preparation. University of Wisconsin-Madison

[13] Pai, Ajit B. and Richard B. Kieburtz, "Global Context Recovery: A New Strategy for Parser Recovery From Syntax Errors," ACM Transactions on Programming Languages and Systems 2, 1, pp. 18-41 (January 1980).

[14] Pennello, Thomas J. and Frank L. DeRemer, "A forward move algorithm for LR error recovery," Fifth ACM Symposium on Principles of Programming Languages, pp. 241-254 (1978).

[15] Ripley, G. David and Frederick C. Druseikis, "A Statistical Analysis of Syntax Errors," Computer Languages 3, pp. 227-240 (1978).

[16] Tai, Kuo Chung, "Syntactic error correction in programming languages," IEEE Trans on Software Engineering SE-4, 5, pp. 414-425 (1978).