A PRACTICAL METHOD FOR SYNTACTIC ERROR DIAGNOSIS AND RECOVERY

by
Michael Burke
Gerald A. Fisher Jr.

Courant Institute
New York University
251 Mercer Street
New York, New York 10012

1.0 GOALS AND FRAMEWORK

Our goal is to develop a practical syntactic error recovery method applicable within the general framework of viable prefix parsing. Our method represents an attempt to accurately diagnose and report all syntax errors without reporting errors that are not actually present. Successful recovery depends upon accurate diagnosis of errors together with sensible "correction" or alteration of the text to put the parse back on track. The issuing of accurate and helpful diagnostics is achieved by indicating the nature of the recovery made for each error encountered. The error recovery is prior to and independent of any semantic analysis of the program. However, the method does not exclude the invocation of semantic actions while parsing or preclude the use of semantic information for error recovery.

The method assumes a framework in which an LR or LL parser, driven by the tables produced by a parser generator, maintains an input symbol buffer, state or prediction stack, and parse stack. The input symbol buffer contains part or all of the sequence of remaining input tokens, including the current token. The LR state stack is analogous to the LL prediction stack; except when restricting our attention to the LL case, prediction stack shall serve as a generic term indicating the LR state or LL prediction stack. The parse stack contains the symbols of the right hand sides that have not yet been reduced.

In most applications a similar stack is required for semantic analysis or for constructing an abstract syntax tree. The parse stack may be combined with such a "semantic" stack with negligible additional cost. When a right hand side has been completed on the parse stack and the next input token can legally be shifted, the right hand side symbols are replaced by the left hand side symbol of the rule being reduced. At the same time a "semantic action" routine may be invoked.

2.0 THE METHOD

2.1 Overiew

The error routine is invoked when no legal parsing action is possible. In such a circumstance the current token is referred to as the error token. The error routine adjusts the input symbol buffer and prediction and parse stacks in such a way that parsing can continue beyond the error token. On entry the parse and prediction stacks are required to be in the configuration that obtained when the token preceding the error token was shifted onto the parse stack. If LR(1) parsing is used without default reductions, this condition will always be satisfied. In an LL, LALR, or SLR implementation, or in the presence of default reductions, however, reductions may occur when the next token is not shiftable. Given the possibility of such premature reductions, it is necessary during the parse to delay reductions until the next token is to be shifted. In LL(1) and LALR(1) implementations, we have developed techniques for deferring reductions without noticable overhead. These techniques are discussed in detail in section 5.1.

The first concern of our method is to determine whether an error is simple: i.e. correctable by a single token

insertion, deletion, or substitution. If so, the correction is made; otherwise a portion of the program text is to be deleted, new text is to be inserted, or both. The discarded text may precede, follow, or surround the error token. The new text inserted, if any, consists of a sequence of tokens inserted to close one or more open scopes. Scopes are syntactically nested constructs such as procedures, blocks, control structures and parenthesized expressions. We refer to this form of recovery as scope recovery. The analysis is divided into two phases called primary recovery and secondary recovery.

2.2 Primary Recovery Phase

The primary recovery phase attempts to make a simple correction or a scope recovery. Its first task is to locate the actual point of error. Starting at the error token, it uses the parse stack to "back up" over the symbols of the uncompleted right hand sides in an effort to identify one of them as the locus of the error. Here backing up over a symbol means removing it from the parse stack and appending it, whether a terminal or nonterminal, to the front of the input symbol buffer. The first symbol on the input buffer is called the current token.

We refer to an attempt in primary recovery to perform a repair at a given point as a trial. The repairs for a trial, in the order attempted, are: token merging, insertion, substitution, scope recovery, and deletion. Token merging is attempted only on the first trial and consists of joining the previous token with the current token or the current token with the next token. Insertion refers to the insertion of a terminal symbol before the current token. Substitution refers to the substution of a terminal symbol for the current token. Substution is quite restricted in that a reserved word is not generally substutited for another reserved word is not substituted for an identifier unless the latter is a "misspelling" of it. Scope recovery is discussed in section 2.4. It is essentially a form of multiple token insertion. Deletion refers to the deletion of the current token. Only terminal symbols are deleted.

In a trial first the set of tokens that will serve as candidates for insertion and substitution is generated by including all tokens that appear to be legal in the current configuration. Each candidate determines a possible repair. The criterion for a successful repair is that it "parse checks": a repair is tested by determining how far it enables the parse to progress into the forward context (up to twenty-five tokens in our implemenation). For token merging and deletion some minimum distance must be achieved. For insertion, substitution, and scope recovery all possible candidate repairs must be considered. When trying a set of candidates, for insertion, say, it may happen that some candidates parse check the full distance. All such candidates are considered valid corrections and are reported as such. For the recovery, however, one is chosen arbitrarily from the set. But it may happen that no candidate parse checks the full distance. If there is exactly one candidate that parse checks the farthest, then that candidate is chosen for the recovery, provided some minimum distance is exceeded. We call this the unique selection criterion. The threshold value is determined by the context in which the check is made, but is at least some implementation-defined minimum value MIN_CHECK (three in our implementation). The minimum amount that must check cannot be too small or spurious corrections may be allowed, and it must at least guarantee that the parser advances after recovery. Because errors may be in close proximity it cannot be too large: choosing the right threshold value is a matter of tuning. To keep it as small as possible, special language specific maps are used to control insertions and substitutions by prohibiting them in certain contexts.

If no correction succeeds at the current token, one or more elements are peeled from the parse stack and appended to the input symbol buffer. The prediction stack is adjusted accordingly. Again attempts at simple correction and scope recovery are made. If this trial also fails, the process repeats, and continues until either a successful correction is made or the backward move down the parse stack arrives at a scope opener. The presence of a scope opener such as "BEGIN" marks the beginning of a grammatical unit that has proven to be flawed. The text is to be repaired within this unit, and so it is unnecessary and undesirable to back up the parse beyond its scope.

2.3 Secondary Recovery Phase

If primary recovery fails, secondary recovery is invoked. The secondary recovery mechanism discards some already parsed and/or future tokens in an attempt to resume the parse. This mechanism may also involve the closing of open scopes. First the parse and predict stacks are restored to the configuration obtaining

upon invocation of the error routine. Starting with the error token, it is checked whether parsing can resume by simply discarding a portion of the parse stack (appropriately adjusting the prediction stack) along with possibly inserting one or more scope closer sequences. A backward move takes place as in primary recovery, but here the parse stack symbols are not returned to the input buffer. We continue backing down the parse stack until either successfully resuming the parse or backing down below the topmost scope opener: as with primary recovery, the presence of a scope opener limits the extent of the backup over left context. We do, however, continue to attempt scope recovery all the way down the parse stack. The length of the parse check performed to test whether parsing can resume is once again dependent upon context, but is at least MIN_CHECK + 2. If the parse cannot be resumed at the current token, it is ignored and the next one obtained and the backup down the parse stack is repeated. Either there is a point at which the parse can continue or the end of file token is reached. In the latter case special action is taken if necessary to ensure recovery.

2.4 Scope Recovery

Primary and secondary recovery include scope completion efforts: i.e. the closing of one or more open syntactic scopes by means of the insertion of appropriate closer token sequences. Typical examples of closer token sequences are the right parenthesis, ")", and "END;". In Ada "END IF;" and "END RECORD;" are further examples of such sequences. This important class of correction stands apart from the others tried during primary recovery in that the closer typically consists of multiple tokens. Nonterminals are not used in closers. The choice of scope openers and closer sequences depends on the language, but is for the most part straightforward. Of course, this mode of recovery may be entirely omitted.

Closing multiple scopes is achieved in the following way. On entry to the parse error procedure the locations of scope openers on the parse stack are determined. Only for these scopes is scope recovery attempted. The set of possible scope closers for a given opener is first determined. Language dependent maps may be used to narrow the possibilities and to prevent premature closing. Each candidate closer sequence is parse checked. If the parse cannot advance through the candidate sequence, the candidate is rejected. If parse advances through the candidate and beyond the threshold distance, the candidate is accepted as the recovery and appended to the input symbol buffer. If the parse advances only through the candidate, then the scope recovery procedure is invoked recursively with the parse stack updated by the parsed candidate to close the next opener. In this way the candidate sequence is extended with additional closer sequences to match other openers. If no sequence of closers is found, the candidate is rejected; otherwise the entire sequence is appended to the input symbol buffer.

3.0 LR PARSING

3.1 The LR Parser

An LR parser, in addition to the token buffer and parse stack, maintains a state stack. Input tokens are shifted onto the parse stack until the handle has been obtained. This determines a rule to be reduced, and its left hand side replaces the handle on the parse stack. The parse stack thus contains the shifted terminal symbols and the reduced nonterminal symbols. An entry for a nonterminal symbol may contain additional information (e.g., a parse derivation subtree, partial abstract syntax tree, or semantic attributes).

3.2 LR Error Recovery

In a primary recovery trial, the set of candidate tokens consists of those that appear to be legal in the current state (we say "appear" because of the possibility that a premature reduction is defined for a given token). Backing up from one trial to the next is simply a matter of peeling the top state and top parse stack element TP from the state and parse stacks, respectively. TP is appended to the token buffer and becomes the current token. Similarly secondary recovery is attempted at a token by peeling one item off the state and parse stacks at a time, at each point checking whether parsing can resume. If the parse cannot be resumed at a token, it is skipped and the next one obtained and the iteration down the state and parse stacks is repeated. Once a state and input are found for which the parse can continue, the state and parse stacks are cut back to this point, and control is returned to the parser.

4.0 LL PARSING

4.1 The LL Parser

LR shift and reduce actions both have LL counterparts. When a terminal symbol on top of the prediction stack matches the next input symbol, it is popped from the stack and the input cursor is advanced. In this case, the terminal is pushed onto the parse stack (the analogue to shifting). A nonterminal on top of the prediction stack is replaced by the RHS of the appropriate rule R as determined by the next input symbol (in our implementation we also push on a rule marker to mark the end of the RHS of a rule). The rule number R is then pushed onto a stack of pending nonterminals: it represents a nonterminal that has been predicted but not yet completed, and so has no LR analogue. As in the LR case, the parse stack consists only of shifted terminals and nonterminals for completed rules. When the right hand side of a rule has been successfully predicted, and the next input symbol can be shifted, then the right hand side symbols are removed from the parse stack and replaced by the left hand side symbol, and the rule number is removed from the pending stack. As in the LR case the parse stack consists of a sequence of symbols representing incomplete right hand sides. In fact the parse stack may be viewed as a sequence of partial right hand sides of the pending rules.

4.2 LL Error Recovery

The set of correction candidates relevant at a given point in primary recovery is determined by the top symbol on the prediction stack. If it is a terminal symbol, then it is the lone candidate; if it is a nonterminal, then the candidate set consists of all terminals for which the given nonterminal can predict a rule. With this set of correction candidtes, simple recoveries are attempted exactly as in the LR case.

If the recoveries fail, then backing up for the next trial is accomplished in straightforward fashion. The topmost pending rules are to be "undone" - i.e. their LHS nonterminals returned to the predict stack - one at a time down through the topmost rule that has some portion of its RHS on the parse stack. The distance between the top of the predict stack and its topmost rule marker indicates how much of the RHS of the topmost pending rule R remains on the predict stack (the rest must be present on the parse stack). The portion of the predict stack above and including the topmost rule is replaced by the left hand side of R. If the entire RHS is on the predict stack, then the new topmost pending rule is processed. Otherwise some of the RHS is on the parse stack; these symbols are removed and appended to the front of the input symbol buffer. In this new configuration the next trial takes place.

From the perspective of the parse stack the LL backup move may be viewed as essentially the same as the LR backup, but with an extra element of constraint introduced. This constraint derives from the superstructure of pending nodes present in the LL derivation tree. Once again backing up entails removing elements from the parse stack and returning them to the input symbol buffer, and continues down the parse stack through the topmost unclosed opener. In this way an entire RHS prefix is removed, not just one symbol of one. In the LR case, all parse stack entries are considered; in the LL case, an entry is only considered as the locus of a trial if it is the left child of its predicted parent. We have found (see section 7.1) that this constraint does not hamper LL recovery. In fact, we are investigating ways to impose efficiently a similar constraint on the LR backup. Errors within a predicted RHS appear very unlikely. In the LR case there is no definite division of the parse stack into a sequence of incomplete right hand sides.

The movement down the parse stack in secondary recovery is performed in the same manner as in the sequence of primary recovery trials. Other than the differences entailed by this, secondary recovery here has the same form as in the LR setting.

5.0 IMPLEMENTATION CONSIDERATIONS

5.1 Deferring Reductions

One quickly discovers that condensation of the left context often results in poor recoveries. Such condensation can result from the use of default reductions or a parsing method weaker than LR. For this reason our recovery algorithm assumes that, upon entry to the error routine, the parse and prediction stacks are in the configuration that obtained when the token preceding the error point was shifted onto the parse stack. Thus premature reductions are not allowed, and so the parser must defer reductions until it is verified that the next token is shiftable. We have implemented an LALR(1) parser that uses default reductions and an LL(1) parser, and in both cases discovered low-cost techniques for accomplishing this.

In our LALR implementation we let the state stack, which (along with the input buffer) determines the parsing actions to be taken, immediately reflect the application of a reduce action when one is defined. However, we do not yet apply the reduction to the parse stack, and instead place the reduce rule number along with the current height of the state stack in a REDUCTIONS queue. When a shift or reduce action is defined, the REDUCTIONS queue is emptied and its deferred reductions applied to the parse stack prior to performing the action. When no action is defined, indicating the presence of an error, the state stack must reassume the configuration that obtained just after the shifting of the previous token. The state stack, then, is to be made to correspond to the current configuration of the parse stack. The minimum-valued element in the list of stack heights indicates the smallest size N assumed by the state stack since the shifting of the previous token. The portion of the state stack below that point already corresponds to the parse stack; the state stack is cut back to this point, and its upper portion is restored by using the parse stack.

In our LL implementation, the presence of a rule marker on top of the predict stack indicates that the topmost pending rule has been completed. Once again, the rule is added to a REDUCTIONS queue in order to defer applying it to the parse stack until it is verified that the next token is shiftable. An index keeps track of the position that the rule most recently completed occupied on the pending stack. In the presence of an error, this index indicates the position at which the non-empty rules in REDUCTIONS are to be returned to the pending stack. Their associated rule markers and the empty rules in REDUCTIONS are returned to the prediction stack, and no other adjustment is required in order to restore the parse to its desired configuration.

5.2 Language Specific Maps

Our method is language-independent, but it does allow for tuning with respect to particular languages and implementations through the setting of language specific maps. Some of these specify the scope opener and closer constructs of the language; others provide the means for controlling primary and secondary recovery actions for certain common or troublesome errors. The recovery algorithm does not depend on the presence of such language specific maps and an implementation may ignore them completely. However, the quality of error diagnosis can be considerably enhanced by careful parameterization, especially in the case of the scope completion maps.

Two maps that prohibit certain primary recovery actions in order to make possible a preferred recovery action are NEVER_INSERT_BETWEEN and NEVER_SUBST_FOR. The map NEVER_INSERT_BETWEEN gives for a token pair (x,y) the set of tokens that should not be inserted between x and y. Thus a possible candidate for insertion is not considered if it is in NEVER_INSERT_BETWEEN(prevtok, curtok), where prevtok is the most recent token on the parse stack and curtok is the first token in the token queue. The map NEVER_SUBST_FOR gives for a token x the set of tokens that it should not be substituted for. For example, [':=', 'BEGIN'] belongs to this set in our Ada implementation; in the case of a missing semicolon before BEGIN this substitution might parse check, but is not desirable.

Scope recovery uses the maps CLOSER_TOKENS and ONLY_CLOSE_BEFORE. The CLOSER_TOKENS map gives for each opener, e.g. 'IF', the set of associated closer tokens, e.g., 'END IF;'. The map ONLY_CLOSE_BEFORE specifies the right context necessary in order for a given closer to be tried as a candidate.

5.3 Diagnostics

Careful attention has been paid the reporting of error recoveries. The diagnostic issued essentially states the repair that effects the recovery. The messages are completely synthesized from the recovery mode and the tokens at the locus of the error. Symbols on the parse stack carry along their token spans expressed in terms of line and column numbers. Examples in the APPENDIX show the messages produced. For secondary recovery, it is often possible to determine that a construct appearing in a list is malformed. For example, if, at the point of recovery, 'statement_list' is the symbol on the parse stack (LR method) or the predict stack (LL method), then the deleted input can be viewed as a malformed statement (most of the time), and so the diagnostic message 'Bad statement' can be issued. When no such message is available, the bad input is simply termed 'Unexpected'.

6.0 COMPARISON WITH OTHER METHODS

Our method builds upon the earlier work of Graham-Haley-Joy [GHJ], Feyock-Lazurus [FL], and Poonen [P]. The recovery methods of [GHJ] and [FL] employ

two levels of recovery in similar fashion to ours: the first attempts a single token repair and the second a deletion of the flawed portion of text based upon identification of an ill-formed program component. Both methods test single token correction candidates by performing a separate forward move for each, but they depart from ours in utilizing semantic information when evaluating a candidate. [GHJ] also assign a deletion and insertion cost to each terminal symbol; the "basic cost" of a repair - "the sum of the costs of the tokens involved" - is another factor considered in the evaluation of a candidate. We have found that our unique selection criterion (section 2.2) and language dependent maps (section 5.2) preclude the need for weighted cost analysis.

Whereas [GHJ] regard any unparsing of left context nonterminals as in violation of the principle that error recovery not incur overhead on correct programs, [FL] back up the parse token by token in an attempt to find the actual error point. The backup halts at the beginning of what is taken to be the relevant program component (the "substructure"), and so the beginning of this unit plays a role similar to that of an opener in our scheme. If the substructure contains more than one error, it is deleted: unlike us, they make no effort to correct errors in close proximity. We do not believe that unparsing nonterminals in the left context incurs overhead on correct programs. Whith our method this option may be used if the parse stack entries for nonterminals contain the derivation tree spanned by that symbol. Is is, of course, very time consuming to unparse the left context. And it complicates the integration of the semantic and parsing phases of a compiler. We have found, however, that unparsing produces no better results, and indeed sometimes produces unwanted secondary recoveries. We find that unparsing is not necessary so long as the grammatical rules defining the syntax of the language are structured so as to prevent the premature condensation of certain symbols before the error point. In the first example in the APPENDIX, the use of FUNCTION instead of PROCEDURE is not discovered until the end of the formal argument part. It is easy to write the grammar rules so that such key terminals stay on the parse stack and are not condensed until the phrase in which they occur has been completed. In our experience such tuning of the grammar adds little or not at all to its parse tables.

Poonen and Johnson [J] base their recovery algorithms upon a stack resynchronization technique that is similar in spirit to our secondary recovery phase. But their methods depend upon the augmentation of the grammar by error productions. Poonen bases resynchronization with forward context upon a single occurrence of one of a set of tokens, rather than upon a sequence of advance tokens. [GHJ] combine Poonen's approach (including the use of error productions) with traditional panic mode in their own secondary recovery. We do not use error productions. They have the advantage of speeding up secondary recovery. But they complicate the grammar and in most cases provide a diagnostic that could just as well be derived automatically from the parse or prediction stack. Also, [GHJ] require that the parser generator "know" about error productions and avoid default reductions when they are used. We place no restrictions on the parser generator and freely use default reductions in the LR case.

The recovery scheme of Penello and DeRemer [PDeR] condenses the right context of the error in a preliminary forward move. This forward move begins by considering all states that may be reached by shifting the token immmediately following the error token. The forward move then parses ahead "in parallel" until sufficient right context has been accumulated. If the forward move does not advance some predetermined amount into the right context, then it is repeatedly applied, starting in each case at the symbol at which it left off, until this amount of forward context is gathered. The correction phase then evaluates single token correction candidates by determining how much of this condensed context can be "consumed" by each repair. When repairs attempted at the detection point fail, each symbol on the parse stack is regarded in turn as the possible erroneous symbol. There is no "unparsing" of nonterminals on the parse stack. If no repair advances the parse even one token, then the first symbol in the forward context is deleted and the entire process repeated.

The preliminary forward move lessens the degree of repetition involved in parse checking individual candidates. However, this approach demands considerable overhead in terms of additional tables required of the paser generator [PDeR]. We find that our implementations spend little time parse checking in primary recovery, where the check is allowed to be long if necessary. Typically, few candidates of a given trial require a check of more than two tokens.

The preliminary forward move also lacks a systematic method for dealing with the presence of errors in the forward context. While the possibility of an

error in the forward context requires us to resort to the heuristic parse distance choice discussed in section 2.2, we have had good success at handling errors in close proximity. Without a systematic method for correcting multiple errors, multiple symbol deletions are of particular importance, and our secondary recovery is designed especially to handle these cases. In [PDeR] a multiple symbol deletion involving a mutilated forward context is accomplished only by means of a costly process that attempts the full gamut of repairs at every symbol before deleting it. Outside of those accounted for by our inclusion of scope recovery within secondary recovery efforts, we have not discovered any cases in which a multiple symbol deletion is appropriately accompanied by a single token insertion. This kind of correction becomes more relevant when nonterminals are allowed as candidates for insertion and substitution. We do not regard the insertion of a nonterminal as desirable, as it would invalidate a semantic action stack or abstract syntax tree. More importantly, it is generally difficult to issue clear and helpful diagnostics accompanying such a correction.

We have found that our scope recovery mechanism, at little cost, significantly enhances many of both our primary and secondary recoveries (see APPENDIX). We are unaware of the incorporation of this form of recovery in any other method.

7.0 EVALUATION OF PERFORMANCE

7.1 Resuts: Pascal And Ada

We have tested our method on the database of erroneous student Pascal programs studied by Ripley and Druseikis[RD]. The same sample data set has been used by [GHJ] and others. Both our LL and LR Pascal implementations diagnose more than 90 percent of the syntax errors in this sample accurately; in no case does either one issue an incorrect or cascaded message. Of the more than 170 recoveries made by each version, in both instances 75% take place in the primary recovery phase. The result of a run of our LL version on more than 75 lines of this sample is included in the Appendix (the LR version yields the same results on these examples). Also included is the result of running the LL version on the same Pascal sample used in [GR], [GHJ], and [PDeR] (again the LR results are identical).

Only in two cases in the [RD] sample do the LL and LR recoveries differ. Both involve secondary recoveries; the difference in both cases pertains to the effect of the structuring of the syntactic rules on the timing of the absorption of terminals into nonterminals on the parse stack. Consider the following one of the two cases:

```
1     PROGRAM P(INPUT,OUTPUT);
2      BEGIN
3         WRITELN(' ',T,' ',LIST[T]);
4      END;
5     END.
```

The error token is the semicolon on line 4, and so the first trial in both versions takes place there. The LR version backs up to the 'END' preceding it for trial two, where deletion of 'END' parse checks and is taken as the correction. In the LL version, trial two takes place at the 'BEGIN' in line two, where the insertion of another 'BEGIN' prior to it is the chosen correction. In this case the LL recovery action is more appropriate, but in another circumstance backing up over so much input at once may result in the loss of an opportunity to make a suitable correction. The relevant rule in the LL grammar is:

```
stmt_part  ::=  BEGIN compound_stmt END
```

If a backup to 'END' is regarded as desirable in the context of this rule, one may introduce the nonterminal 'end' that derives the terminal 'END', and include it instead on the RHS. This change is not sufficient, however, because the correct parsing of the END token has forced the reduction to compound_stmt.

We have also tested our method on erroneous Ada programs. One particular test program of slightly more than 100 lines includes 50 syntax errors; the two versions perform identically, accurately identifying all errors. Part of this sample is included in the APPENDIX.

7.2 Efficiency

We implemented the LL and LR parsers as separate parse modules attachable to a translator writing system, all written in the very high level language SETL. The programs were executed on a VAX 11/780. The LL parser processes the 725 line Pascal sample in 35 minutes and 22 seconds of CPU time. The LR version spends 28 minutes and 20 seconds on the same sample. The same program, with errors corrected, can be parsed in 15 minutes and 36 seconds using the LL parser and 7 minutes 35 seconds using the LR parser. Subtracting the parsing time from the total time and dividing by the number of errors gives an average time per error of 7 seconds for

both methods. The very large execution times stem from the use of SETL. Our experience with recoding SETL programs into a lower level language, such as PL/I, suggests that a speed up factor of 30 or more is readily achieved. Thus the average time to recover from an error would be about 0.25 seconds. We were surprised to find that LL parsing is more than twice as slow as LR parsing: this may be an artifact of using SETL. We did observe that with LL error recovery the candidate sets are appreciably smaller and there are fewer trials than with LR recovery. However, the average time spent parse checking a candidate is significantly greater.

8.0 SHORTCOMINGS AND FUTURE WORK

A fundamental limitation of our method lies in the choice of the first correction candidate that successfully parse checks as the suitable correction. This approach is obviously advantageous with respect to time efficiency, and makes it unnecessary to record and compare parse check outcomes. But the order in which the types of simple correction are attempted is then relevant. Through experimentation we have discovered an order in which the most suitable correction is seldom precluded by a less desirable one that happens to be tried earlier. But of course there is no order that universally prevents such an occurrence, and many of the entries in NEVER_SUBST_FOR and NEVER_INSERT_BETWEEN (see section 5.2) are included to suppress a correction that would otherwise take place before a preferred correction is tried.

Taking the first candidate that checks implies a bias not only within but also across trials. If in the first trial a candidate succeeds, then a second trial is never attempted. Since trial one attempts correction at the at the error token and each succeeding trial takes place to the left of the previous one, our corrective actions have a builtin left to right bias. One may opt for a method that checks all candidates over a single trial or over all trials before deciding which (if any) applies, and we are experimenting with this approach.

Another shortcoming of our method concerns the advance token parse check itself. An implicit assumption of our technique is that although the left context of the error detection point has proven to contain an error, the right context is sufficiently error free to guide the error recovery process. In judging correction candidates in terms of their agreement with the forward context, the method has little safeguard against the possibility that an additional error occurs near the one already detected. We make the parse check as small as possible to allow for the separate diagnosis and correction of errors in close proximity, but the possibility remains that the errors are too close to allow a successful parse check by any candidate for correction of the first error. The result in this case is that the portion of the text containing both errors is deleted in secondary recovery.

A solution to the problem of arbitrarily dense error occurrences is to invoke recursively the error recovery routine when a parse check blocks, and associate with each correction candidate the number of errors that its choice would imply correcting in the forward context. Such an approach would reduce the importance of those language specific maps whose role is to reduce the likelihood of a spurious correction. This method is under investigation.

9.0 REFERENCES

[FL] Feyock, S., Lazurus, P., "Syntax-directed Correction of Syntax Errors", Software Practice and Experience, Vol. 6, 1976.

[GHJ] Graham, S.L., Haley, C.B., Joy, W.N., "Practical LR Error Recovery", SIGPLAN Notices, August 1979.

[GR] Graham, S.L., Rhodes, S.P., "Practical Syntactic Error Recovery", CACM, November 1975.

[J] Johnson, S.C., YACC - Yet Another Compiler Compiler. Bell Laboratories, Murray Hill, 1977.

[P] Poonen, G., "Error Recovery For LR(k) Parsers", Information Processing, August 1977.

[PDeR] Pennello, T.J. and DeRemer, F.A., "A Forward Move for LR Error Recovery", Conf. Record ACM Symposium on Principles of Prog. Lang., January 1978.

[RD] Ripley, G.D., Druseikis, F.C., "A Statistical Analysis of Syntax Errors", Journal of Computer Languages, Vol. 3, 1978.

Acknowledgement

This work was supported by Army Contract DAAK80-81-K-0071, Fort Monmouth, NJ.

```
    { A SAMPLE FROM THE [RD] COLLECTION OF ERRONEOUS PASCAL PROGRAMS }

    1   PROGRAM P(INPUT,OUTPUT);
    2       FUNCTION TOPSORT(VAR X : ORDER, VAR Y : SORTED, X:INTEGER);
                                    ^                       ^
*** Syntax Error: ";" expected instead of ","
*** Syntax Error: ";" expected instead of ","
                  ^

*** Syntax Error: "PROCEDURE" expected instead of "FUNCTION"
    3       BEGIN
    4       END;
    5   BEGIN
    6       X:=1
    7   END.
    8
    9   PROGRAM P(INPUT,OUTPUT);
   10   BEGIN
   11       IF X:= 0 THEN X:=1
                ^

*** Syntax Error: "=" expected instead of ":="
   12   END.
   13
   14   PROGRAM P(INPUT,OUTPUT);
   15       VAR I:REAL;
            <---------->
*** Syntax Error: Unexpected input
   16       TYPE ORDER = ARRAY[1..MAXRELATIONS,1..2] OF INTEGER ;
   17       VAR Q:INTEGER;
   18   BEGIN
   19   END.
   20
   21   PROGRAM P(INPUT,OUTPUT);
   22       VAR L,N: REAL;
   23       VAR X,NONPRIME,PRIME: INTEGER;
                ^

*** Syntax Error: Unexpected "VAR" ignored
   24   BEGIN
   25   END.
   26
   27   PROGRAM P(INPUT,OUTPUT);
   28   BEGIN
   29       WRITELN(' ',9,'X',10,'M',9,'X]',9,'APPROX X]',19,
                       ^   ^    ^   ^      ^

*** Syntax Error: "," expected instead of ";"
*** Syntax Error: "," expected instead of ";"
*** Syntax Error: "," expected instead of ";"
*** Syntax Error: "," expected instead of ";"
*** Syntax Error: "," expected instead of ";"
*** Syntax Error: ")" expected instead of ","
   30   END.
   31
   32   PROGRAM P(INPUT,OUTPUT);
   33   BEGIN
   34       FOR I := 1 STEP 1  UNTIL LISTSIZE - 1 DO
            <----------^^^^------------------------>
*** Syntax Error: Bad statement
   35           X:=1
   36   END.
   37
   38   PROGRAM P(INPUT,OUTPUT);
   39       LABEL 1,999;
   40   1: BEGIN
            ^

*** Syntax Error: "BEGIN" expected before this token
   41       READ(N);
   42   END.
            ^

*** Syntax Error: "END" expected after this token
   43
```

```
   44    PROGRAM P(INPUT,OUTPUT);
   45        FUNCTION FOO(VAR X: ARRAY[1..MAX] OF INTEGER): INTEGER;
                                 <-------------->
*** Syntax Error: Unexpected input
   46        VAR Q:INTEGER;
             ^

*** Syntax Error: statement part missing for PROCEDURE or FUNCTION on line 45
   47    BEGIN
   48    END.
   49
   50    PROGRAM P(INPUT,OUTPUT);
   51    BEGIN
   52        IF NON PUSH(I) THEN X:=1
             ^

*** Syntax Error: Reserved word "NOT" misspelled
   53    END.
   54
   55    PROGRAM P(INPUT,OUTPUT);
   56    BEGIN
   57        IF COUNT[LISTDATA[SUB] := 0 THEN
                                    <-->
*** Syntax Error: Unexpected input -- "]" expected after this token on line 57
   58            BEGIN
   59                F := LISTDATA[SUB];
   60             END;
   61    END.
   62
   63    PROGRAM P(INPUT,OUTPUT);
   64        FUNCTION FACTORIAL( VAR: X: INTEGER): INTEGER]
                                  ^                   ^

*** Syntax Error: Unexpected ":" ignored
*** Syntax Error: ";" expected instead of "]"
   65            VAR Q:INTEGER;
   66        BEGIN
   67        END;
   68    BEGIN
   69    END.
   70
   71    PROGRAM P(INPUT,OUTPUT);
   72    BEGIN
   73        REPEAT
   74            WRITELN(' ------------------');
   75        UNTILL EOF(INPUT);
                 ^

*** Syntax Error: Reserved word "UNTIL" misspelled
   76        X:=1
   77    END.
   78

  22 parse errors detected
  Parsing time: 167 seconds
```

```
    {  PASCAL EXAMPLE THAT APPEARS IN [GR], [PDeR], AND [GHJ]  }

    1   program cacm(input,output);
    2   label 1,2,3;
    3   var a,b: array[1..5 1..10] of integer;

*** Syntax Error: "," expected after this token
    4       i,j,k: integer;
    5   begin
    6       3:  i + j > k + 1*4 then go 1 else k is 2;

*** Syntax Error: "IF" expected after this token
*** Syntax Error: Reserved word "GOTO" misspelled
*** Syntax Error: ":=" expected instead of "IS"
    7       a 1, 2 := b[3 * ( i + 4, j*/k]

*** Syntax Error: "[" expected after this token
*** Syntax Error: "]" expected after this token
*** Syntax Error: ")" expected after this token
*** Syntax Error: "IDENTIFIER" expected after this token
*** Syntax Error: ";" expected after this token
    8       if i=1 then then goto 3;

*** Syntax Error: Unexpected "THEN" ignored
    9   2: end.

  10 parse errors detected
  Parsing time: 46 seconds


--  Ada EXAMPLE.  INCLUDES SCOPE RECOVERY AND SECONDARY RECOVERY

    1   program etests is

*** Syntax Error: "PROCEDURE" expected instead of "PROGRAM"
    2
    3       j, k, l, : integer;

*** Syntax Error: Unexpected "," ignored
    4
    5       a: array (INTEGER range 1..10) is integer;

*** Syntax Error: "OF" expected instead of "IS"
    6
    7       type b is INTEGER range 1..30;

*** Syntax Error: Unexpected "INTEGER" ignored
    8
    9       proc count is

*** Syntax Error: Reserved word "PROCEDURE" abbreviated
   10           x: integer;
   11            GET(x);

*** Syntax Error: "BEGIN" expected before this token
   12           PUT(x)

*** Syntax Error: ";" expected after this token
   13       end count;
   14
   15       procedure q is seperate;

*** Syntax Error: Reserved word "SEPARATE" misspelled
   16
```

```
   17       function DAYS_IN_MONTH(M: MONTH IS_LEAP: BOOLEAN) return DAY is
                                      ^
*** Syntax Error: ";" expected after this token
   18       begin
   19          case M of
                      ^
*** Syntax Error: "IS" expected instead of "OF"
   20             when SEP | APR | JUN | | NOV => return 30;
                                        ^
*** Syntax Error: Unexpected "|" ignored
   21                   FEB => return 28;
                        ^
*** Syntax Error: "WHEN" expected before this token
   22             whan APR => return 30;
                   ^
*** Syntax Error: Reserved word "WHEN" misspelled
   23                   others => return 31;
                        ^
*** Syntax Error: "WHEN" expected before this token
   24          end case;
   25
   26          z(y - 5*j + k rem 7) then
               ^
*** Syntax Error: "IF" expected before this token
   27             x := x + 1;
   28             go to label;
                  <--->
*** Syntax Error: "GOTO" expected instead of "GO" "TO"
   29          end if;
   30
   31          loop
   32              if x > 0 then y := 2;
   33              if y < 0 then z := 3;
                                       ^
*** Syntax Error:  "END IF;" inserted to match "IF" on line 33
                                          ^
*** Syntax Error:  "END IF;" inserted to match "IF" on line 32
                                          ^
*** Syntax Error:  "END LOOP;" inserted to match "LOOP" on line 31
                                          ^
*** Syntax Error: "END;" inserted to match "BEGIN" on line 18
                                          ^
*** Syntax Error: statement part missing for unit starting on line 17

  21 parse errors detected
  Parsing time: 120 seconds
```