A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery

MICHAEL G. BURKE and GERALD A. FISHER
Thomas J. Watson Research Center

This paper presents a powerful, practical, and essentially language-independent syntactic error diagnosis and recovery method that is applicable within the frameworks of LR and LL parsing. The method generally issues accurate diagnoses even where multiple errors occur within close proximity, yet seldom issues spurious error messages. It employs a new technique, parse action deferral, that allows the most appropriate recovery in cases where this would ordinarily be precluded by late detection of the error. The method is practical in that it does not impose substantial space or time overhead on the parsing of correct programs, and in that its time efficiency in processing an error allows for its incorporation in a production compiler. The method is language independent, but it does allow for tuning with respect to particular languages and implementations through the setting of language-specific parameters.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*user interfaces*; D.2.6 [**Software Engineering**]: Programming Environments; D.3.4 [**Programming Languages**]: Processors—*compilers*; *parsing*; *translator writing systems and compiler generators*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: LL parser, LR parser, syntactic error diagnosis, syntactic error recovery, syntactic error repair

1. INTRODUCTION

This paper presents a powerful, practical, and essentially language-independent syntactic error recovery method that is applicable within the frameworks of LR and LL parsing. An error recovery method is powerful insofar as it accurately diagnoses and reports all syntactic errors without reporting errors that are not actually present. A successful recovery, then, has two components: (1) an accurate diagnosis of the error, and (2) a recovery action that modifies the text in such a way as to make possible the diagnosis of any errors occurring in its right context. An "accurate" diagnosis is one that results in a recovery action that effects the "correction" that a knowledgeable human reader would choose. This notion of accuracy agrees with our intuition but cannot be precisely defined. In some instances, of course, the nature of the error is ambiguous, but at the very least, the diagnosis and corresponding recovery should not result in an excessive

Authors' address: Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

deletion of tokens or spurious or missed error detections. The development of a minimum-distance corrector [1] is not the purpose here, although in practice a minimum-distance correction should almost always be chosen.

The "practicality" requirement imposes certain constraints: Substantial space or time overhead, in terms of the parsing framework or enhancements of the grammar, should not be incurred. Thus the time and space costs of parsing a correct program should not appreciably increase. It is further required that in practice the average time cost of a recovery should not vary with program length. Also, this cost should be small enough to allow for incorporation of the method in a production compiler.

Our method is language independent, but it does allow for tuning with respect to particular languages and implementations through the setting of language-specific parameters. Some of these provide the means for heuristically controlling recovery actions for certain common or troublesome errors; others improve recoveries for errors involving absent or distorted scope information. The method does not depend on the presence of these parameters, and an implementation may ignore them completely.

Our method is described in Section 2. Section 2.1.3 motivates and describes the technique of parse action deferral. In Section 3 we consider implementation issues pertaining to time and space efficiency, language-dependent tuning, and diagnostic messages. Section 4 gives the empirical measurements on which we base our claim of having developed a powerful and practical method. Section 5 summarizes our results and compares our method with earlier efforts that have influenced our work.

2. THE METHOD

2.1 Overview

2.1.1 *The Parsing Framework.* The method assumes a framework in which an LR or LL parser maintains an input token buffer TOKENS, a state or prediction stack, and a parse stack. The *parse configuration* thus has three components: the configuration of TOKENS, that of the state or prediction stack, and that of the parse stack. TOKENS is a queue containing part or all of the sequence of remaining input tokens. The *current token*, denoted CURTOK, is the front element of TOKENS. The token immediately preceding CURTOK in the source program shall be denoted as PREVTOK.

The LR state stack and the LL prediction stack are analogous, and our method can be applied with essentially equal ease and effectiveness in the presence of either. For clarity, our discussion will focus primarily on the LR version of our method. The LL version and the differences between the LR and LL versions are described in [3].

The parse stack consists of a sequence of recognized nonterminal and terminal symbols. A recognized nonterminal is one that forms the left-hand side of a rule whose right-hand side has been completed, and a recognized terminal is one that has been shifted. In our notation here, a parse stack configuration is represented as a sequence of terminal and nonterminal symbols, enclosed in brackets. The first entry in the sequence is the "bottom" item of the stack. Another way of

viewing the parse stack is that it 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). The application of a reduction to the parse stack consists in the replacement of the symbols belonging to the completed right-hand side by the left-hand side symbol of the reduce rule; at the same time, a "semantic action" routine may be invoked. We make no assumptions about whether semantic actions accompany reduce actions in this manner or occur in a separate pass. In the latter case, at least a tree-build or output action must occur.

2.1.2 *The Three Phases of Recovery.* The error recovery procedure 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 parse configuration so as to allow the parse to advance at least a single token beyond the error token. A fundamental determination of the routine is whether the error is to be repaired by a single token modification of the source text. This modification may take the form of the insertion or deletion of a single token, the substitution of one token for another, or the merging of two tokens into one. Such a single token repair shall be referred to as a *simple repair*, and an error repaired in this manner as a *simple error.* If the error is not simple, then a portion of the program text is to be deleted, new text is to be inserted, or both. 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.* A *secondary recovery* consists in discarding text that precedes, follows, or surrounds the error token. The analysis is thus divided into simple, scope, and secondary recovery.

The determination of whether an error is simple divides into two separate problems: the generation of the set of simple repair candidates, and the selection of a single candidate (or rejection of all the candidates) in this set. Our solution to the selection problem is detailed in Sections 2.2.3 and 3.2.1. Our approach to the candidate generation problem rests upon the technique of deferring parse actions. We now motivate our introduction of this technique by considering the difficulties posed by this problem.

2.1.3 *Deferring Parse Actions.* The error token is not necessarily the token that is in error. Consider the following Pascal declaration:

(P.1)
PROCEDURE *FACTORIAL*(*X*: INTEGER; VAR *FACT*: INTEGER): INTEGER;

In this example (drawn from the sample in [16]) the keyword "PROCEDURE" is used where "FUNCTION" seems to be intended. Whatever the intention of the programmer, a simple repair can be effected by substituting "FUNCTION" for "PROCEDURE." The error token is the colon following the right parenthesis. In this case, then, the point of error detection follows the occurrence of the error by 12 tokens. One may assume that the prefix (i.e., the portion of the program preceding the error token) is good and let simple recovery efforts fail in a case

such as this one. Recovery then consists of discarding the error token and tokens immediately succeeding it in the right context until the prefix can be extended by the remaining text. In this case the discarding of the error token (the colon) and the token following it ("INTEGER") would allow the prefix to be extended and result in a reasonable recovery. But this approach sometimes results in the deletion of many tokens of right context where a simple repair would suffice; it is also possible that the prefix cannot be extended by the right context at all without deleting some portion of the prefix itself. To undertake simple recovery in a generally effective manner, the possibility that the prefix is not correct must be taken into account.

By backing down the parse stack and considering the possible simple repairs at each of its elements, one can effect a simple repair at a point in the prefix. The hope is that the erroneous tokens are still present on the parse stack, but it cannot in general be guaranteed that an erroneous token will not have been absorbed into a nonterminal before detection of the error. Also, undesirable reduce actions may have been induced by its presence, as in Example P.2, also drawn from the sample in [16].

(P.2)

```
 1  PROGRAM P;
 2    VAR X: INTEGER;
 3  BEGIN
 4    X := 20;
 5    IF X = 1 THEN
 6      IF X = 1 THEN
 7        X := 1
 8      ELSE
 9        WRITELN(' ');
10    ELSE
11      X := 2
12  END.
```

The error token is the "ELSE" that follows the semicolon. Prior to shifting the semicolon, the sequence of tokens succeeding "BEGIN" and preceding the semicolon (given a typical grammar for Pascal) is reduced to [*stmt_list* ";" IF *expression* THEN *statement* ELSE *statement*], then to [*stmt_list* ";" *statement*], and finally to [*stmt_list*]. The conditional statement has been absorbed into the *stmt_list*. The semicolon is then shifted. Thus, upon entry to the error recovery routine, the parse stack contains the suffix [*stmt_list* ";"]. Deleting the error token "ELSE" on line 10 renders a syntactically correct program, but deleting the semicolon is more likely to yield what the programmer intended: The role in Pascal of the semicolon as a statement separator rather than a statement terminator is a common source of confusion. It is not our aim to guess the intent of the programmer, but this is the repair that a human reader would choose in that it best leaves the sense of the program intact. In any case it is the generation of repair candidates, rather than the selection process, that is under consideration here. The semicolon is still present on the parse stack, so deletion of it should at least be recognizable as a viable repair candidate. But this deletion will not at this point result in a correct program, because the nonterminal *stmt_list* cannot legally be followed by the symbol "ELSE."

In an LL, LALR, or SLR implementation, or in an LR implementation that utilizes default reductions, reductions may occur when the current token is not shiftable. In these contexts, then, the erroneous token may induce undesirable reduce actions even if it is not shifted. In such a case, the erroneous token is the error token, but late detection has occurred in that the prefix (as represented by the parse stack) is incorrect.

The difficulties posed by unwanted reduce actions can be accommodated by "unparsing" while backing down the parse stack—that is, undoing reduce actions and recovering the terminal symbols spanned by nonterminals on the parse stack. But unparsing is time consuming, and a full unparsing mechanism requires that a full derivation tree be maintained on the parse stack. Building this syntax tree would add to both the space and time cost of parsing a correct program. Where syntactic and semantic analyses are performed in a single pass, unparsing would also require the introduction of a costly mechanism for "undoing" semantic actions.

The effect of some limited degree of unparsing can be achieved, however, by deferring the application of shift and reduce actions to the parse stack. This approach is compatible with generating an abstract-syntax tree and/or performing semantic actions as parse stack reductions occur. In [4] we have already described, within both the LR and LL frameworks, mechanisms for deferring parse stack reductions until a shift is about to occur. These mechanisms were specifically developed as a solution to the problem of premature reductions. But they may be understood as unparsing mechanisms in that they allow the parse to be restored to the configuration that obtained after shifting the token previous to the error token.

Regarding this mechanism as a one-token deferral of reduce actions, it may be generalized to a k-token deferral mechanism. That is, the deferring of a single sequence of reduce actions may be generalized to the deferring of k sequences of reduce actions and the $k - 1$ shift actions occurring between them. The tokens for which shift actions have been deferred shall be referred to as *deferred tokens*.

Let us reconsider Example P.2, supposing that a two-token deferral mechanism is in effect (i.e., two sequences of reduce actions and the shift occurring between them are deferred). In this case the most recent action applied to the parse stack at the point of error detection has been the shifting of the right parenthesis on line 9. If the parse stack is used to restore the state stack to the configuration corresponding to this point of the parse, then the undesired reductions would in effect not have taken place, and so the desired degree of unparsing is achieved.

Token deferral may also be viewed as double parsing. One parser simply checks for syntactic correctness and performs no real reduce actions. The second parser is always $k - 1$ tokens behind, always has correct input, and performs reduce actions on the parse stack. In our implementation the deferred tokens and sequences of reductions are maintained in a deferred tokens queue and a deferred rules queue, respectively.

We regard the generation of simple repair candidates at points in the left context of the error token, then, as having two dimensions: backing down the parse stack and deferring parse actions. But recall the constraint that no appreciable overhead be incurred for the parsing of correct programs. The degree

to which deferring parse actions incurs overhead is quantitatively described in Section 4.2. The measurements given indicate the degree to which we have succeeded in developing a high-quality, low-cost error recovery method. Of course, it is ultimately the specific needs of a particular implementation that determine whether the overhead is "appreciable" or even prohibitive. In Sections 2 and 3.1.3, we describe versions of our method that limit its general form by excluding one or both dimensions of simple candidate generation. Measurements given in Section 4.2 provide an accurate description of the performance and efficiency trade-offs that obtain between these different versions. Our fundamental claim is that the method offers an implementor a range of choices, one of whose endpoints is excellent performance with reasonable efficiency, and the other, reasonable performance with excellent efficiency.

2.2 Simple Recovery

2.2.1 *The Parse Configuration.* As stated in Section 2.1.1, an LR parse configuration has three components: a sequence of tokens [TOK_1 = CURTOK TOK_2 ... TOK_{LAST}], a state stack [SS_1SS_2 ... SS_{TOP}], and a parse stack [PS_1PS_2 ... PS_{TOP}]. Additionally, where the level of parse action deferral is $k + 1$ (Section 2.1.3), our parser maintains a sequence of deferred tokens [$DT_1DT_2 \ldots DT_k$] for which state but not parse stack actions have been applied. (The deferred token buffer does not always contain the full k tokens: At the start of the parse, it is empty, and we shall see that it is emptied when a syntactic error is detected.) When a syntactic error is detected at CURTOK (which is then regarded as the error token ET for this recovery), the sequence of reduce actions immediately preceding the shifting of DT_1 has not yet been applied to the parse stack (thus the element PS_{TOP} is a terminal symbol). The state stack can then be regenerated to correspond to the parse stack, in effect "unparsing" to that point.

We now introduce terminology that will enable a precise statement of the simple recovery phase of our algorithm. For expository purposes, here we regard the parse stack and deferred token sequence as concatenated into the single tuple

$$[LC_1 = PS_1 \ldots LC_{TOP} = PS_{TOP}LC_{(TOP+1)} = DT_1 \ldots LC_{(TOP+k)} = DT_k].$$

We denote this tuple as LEFT_CONTEXT: It represents the portion of the program preceding the error token. We define the left context point LCP_s, corresponding to an element LC_s, of LEFT_CONTEXT as the point in the parse immediately following the shifting of $LC_{(s-1)}$ (a nonterminal symbol NT is thought of as being "shifted" here when *GOTO*(NT, SS_{TOP}) is applied, yielding a value that is pushed onto the state stack).

A *simple recovery trial* is a testing of all the simple repairs possible at a given left context point LCP_T (we refer to this trial as the one taking place at T). The most straightforward approach to simple recovery would be to perform a trial at each parse stack element and deferred token, as well as the error token. The number of trials that are in fact performed is a question of efficiency. We have developed a technique, described in Section 3.1.1, that limits the number of trials by recognizing when it is impossible for a trial to the left of some point in the left context to yield a successful simple repair.

In general, then, in simple recovery a trial takes place at the error token and at each of a consecutive sequence of one or more (possibly all) deferred tokens immediately preceding it. If a trial is held at every deferred token, then in the *general version* of our method a trial is also held for each element of some (possibly empty) suffix of the parse stack. In the *deferred version*, trials are held only at deferred tokens (none are held at parse stack elements). The tokens and parse stack elements at which trials are to take place shall be referred to as *trial tokens* (in the general version, a trial "token" may be a nonterminal, as it may be a parse stack element).

For an efficient progression from one trial to the next, in our implementation trials are performed in succession from the leftmost trial point LC_L to the rightmost (the error token). Prior to performing the first trial, the parse configuration is set to correspond to the point LCP_L. If LC_L is a parse stack element, then the suffix $[PS_L \ldots PS_{TOP}]$ is removed from the parse stack (see Figure 1a); if LC_L is a deferred token, then the actions associated with the sequence $[DT_1 \ldots DT_m = LC_{(L-1)}]$ are applied to the parse stack (see Figure 1b).

In either case the state stack is then rebuilt to correspond to the parse stack. In the course of this generation, a terminal symbol is parsed as always; a nonterminal symbol is simply shifted by applying $GOTO(NT, SS_{TOP})$ as indicated above (we assume that a nonterminal symbol can be recognized as such). The suffix of LEFT_CONTEXT beginning with LC_L is then concatenated to the front of TOKENS (thus the deferred token queue has been emptied). This is the parse configuration as the first trial commences. It may remain intact until the error repair is chosen and applied (or until secondary recovery commences), since only the state stack and TOKENS are needed in testing simple and scope repair candidates, and STATE_STACK_COPY and TOKENS_COPY can be used for this purpose (after being set to the state stack and TOKENS, respectively, prior to the first simple recovery trial). For clarity, we assume that these copies are used during the repair trials, but this is an implementation choice: The original structures could be manipulated instead, so long as they are restored appropriately prior to the application of the chosen simple or scope repair or to secondary recovery.

2.2.2 *The Simple Recovery Trials.* We now consider the trial taking place at LC_s. Simple recovery has advanced from the trial at $LC_{(s-1)}$ to this trial by removing $LC_{(s-1)}$ from TOKENS_COPY and applying the sequence of reduce actions immediately preceding LC_s, as well as the shifting of LC_s, to STATE_STACK_COPY. At the beginning of this trial, the configuration of TOKENS_COPY is $[LC_s \ldots ET \ldots TOK_{LAST}]$. In the course of the trial, a set of repair candidates is generated, including token insertions prior to the current trial token LC_s, token substitutions for LC_s, and deletion of LC_s. (If LC_s is a nonterminal, we do not generate its deletion or any substitutions for it as repair candidates). Each terminal symbol is considered as a candidate for both insertion and substitution. Where a terminal symbol T is considered as a candidate for insertion, T is appended to the front of TOKENS_COPY, resulting in $[T\ LC_s, \ldots TOK_{LAST}]$. Where T is considered for substitution, it replaces LC_s in TOKENS_COPY, resulting in $[T\ LC_{(s+1)} \ldots TOK_{LAST}]$. A substitution is regarded as a misspelling candidate if it indicates the substitution of a

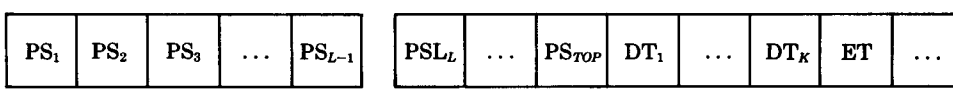

Fig. 1a. First trial configuration, where the leftmost trial token was a parse stack element upon entrance to error recovery. The state stack configuration corresponds to that of the parse stack.

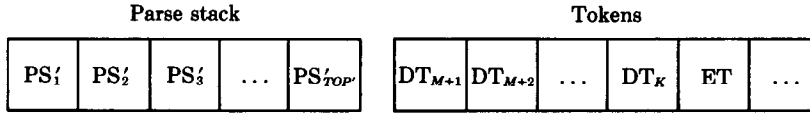

Fig. 1b. First trial configuration, where the leftmost trial token was a deferred token upon entrance to error recovery. The parse and state stacks have been advanced to configurations corresponding to the left context point of DT_{M+1}.

reserved word for an identifier and the reserved word and identifier pass a string proximity test. For the deletion candidate, TOKENS_COPY is set to $[LC_{(s+1)} \ldots TOK_{LAST}]$. The other simple repair mode is merging. In the trials that take place at the error token and at the token immediately preceding it, the merging of the current trial token with the next token is tried if such a merge yields a keyword of the language (e.g., "GO" and "TO" may be merged to "GOTO").

Given the TOKENS_COPY configuration corresponding to a candidate, and the STATE_STACK_COPY configuration corresponding to the current trial, a parse check determines how many tokens in the right context of the error token can be consumed before blocking on an error. A candidate must allow the parse to advance at least MIN_ADVANCE (one in our implementation) tokens into the right context of the error token to remain in consideration. It is the task of the repair trials to generate the set CANDIDATES of those candidates that parse check the farthest distance into the right context.

The fundamental criterion by which a candidate is judged, then, is the distance it enables the parse to advance without blocking on an error. The parse check criterion is a simple one, and yet we have found that, combined with the selection process described below, it achieves the end of generally choosing the most appropriate repair. Its effectiveness is reflected in the measurements provided in Section 4.1.

2.2.3 *Evaluation of Simple Repair Candidates.* At the conclusion of the simple recovery trials, CANDIDATES is evaluated to determine whether a simple repair is to take place. First it may be pruned through the application of heuristic criteria, such as the ones we describe in Section 3.2.1. We have arrived at these criteria largely through experimentation with erroneous Pascal and Ada[1] programs, developing general strategies by identifying the general character of particular examples.

After CANDIDATES has been pruned, the selection of a candidate (or rejection of all candidates) takes place. If one or more repair modes have a single remaining

[1]Ada [2] is a registered trademark of the United States Government (Ada Joint Program Office).

candidate, then a candidate from one of these is chosen using the heuristic preference order: merge, misspellng, insertion, deletion, substitution. The success of a merge or misspelling is unlikely to be accidental, and so these are given top preference. A substitution involves inserting one token and deleting another, and so is the least desirable mode of repair insofar as the goal is to alter the original text as little as possible. We found that, where both an insertion and a deletion succeed, the insertion is usually more appropriate. This ordering, like the other heuristic criteria we employ, was arrived at largely through experimenting with our error samples. Apart from the natural preference given to merge and misspell candidates, it should not be construed as a hard and fast order and may be tuned to suit an implementor's taste. In any case, such tuning has very little effect on the performance of our method (see Section 3.2.2).

If no mode has a single candidate but the farthest parse check distance equals or exceeds a threshold value (MIN_ADVANCE + 3 in our implementation), then a mode is chosen by applying the same preference order as above. In this case the chosen mode has more than one candidate: One is chosen arbitrarily as the repair, but all are reported in the diagnostic. If neither of the above conditions is met, then no simple repair is made.

We illustrate simple recovery by tracing the action of our error recovery routine, with the level of deferral set to two, on Example P.2 of Section 2.1.3. The parse error is detected at "ELSE" on line 10, and so the deferred token buffer contains the preceding semicolon. The number of trials to be held is determined to be two. The first trial is attempted at the deferred semicolon: No insertion, substitution, or merge candidate parse checks successfully into the right context. Deletion of the semicolon parse checks through the end of the program. The second trial is attempted at the error token "ELSE." No insertion or merge candidate checks into the right context. The substitution of "BEGIN" for "ELSE" checks four advance tokens (through "END"); the substitution of "REPEAT" and "FOR" for "ELSE" checks three advance tokens (through "2"); substituting the semicolon for "ELSE" and deleting "ELSE" check all the way. Thus, after all simple recovery trials have been conducted, three candidates remain: the substitution of the semicolon for "ELSE," the deletion of the semicolon, and the deletion of "ELSE." Preference criteria are then applied in accordance with the heuristic rules described in Section 3.2, and deleting the semicolon is the chosen candidate (the other two candidates are disfavored because they involve the deletion of a keyword). This simple recovery, then, results in the following diagnostic:

```
 1  PROGRAM P;
 2    VAR X: INTEGER;
 3  BEGIN
 4    X := 20;
 5    IF X = 1 THEN
 6      IF X = 1 THEN
 7        X := 1
 8      ELSE
 9        WRITELN(' ');
                      ↑
*** Syntax Error: Unexpected ";" ignored
10    ELSE
11      X := 2
12  END.
```

2.3 Scope Recovery

The scope recovery phase only takes place if simple recovery efforts fail. This mode of repair stands apart from the simple recovery modes in that it can and typically does involve a multiple insertion of tokens. This phase involves scope completion efforts: that is, 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. In an implementation they are specified for the given language by the set CLOSERS. In that the choice of scope closer sequences used to control scope recovery is language dependent, this phase of our algorithm (unlike simple and secondary recovery) cannot be performed without the specification of language-dependent parameters. Insofar as this specification is optional, so is this entire phase. However, the specification of scope closer token sequences for a particular language is straightforward, and in the case of a block structured and syntactically complex language such as Ada, scope recovery significantly enhances the performance of our algorithm on errors that result in absent or distorted scope information.

In this section we confine the discussion to what our scope recovery algorithm essentially entails. This algorithm assumes a somewhat more complicated form—described in Section 3.1.2—as we implemented it, but the additional features described there pertain only to its time efficiency. A *scope recovery trial* is held at each point where a simple recovery trial takes place. Successful scope recovery generally occurs at or near the error token, but not always. The advance from one trial to the next takes place in the same manner as with simple recovery.

Each closer sequence is parse checked as a multiple token insertion just prior to the current token. If the parse cannot advance through the candidate sequence, the candidate is rejected. If the parse advances through the candidate and a token beyond the error token, the candidate is accepted as the recovery. Where the parse advances through the candidate but not far enough for acceptance, the scope recovery procedure is invoked recursively in an effort to close multiple openers. The candidate sequence is extended in turn by each closer sequence, and each new candidate is then processed in the same manner as above. Eventually either all candidates fail, or one succeeds and recovery consists of insertion of the entire sequence.

We illustrate our scope recovery technique with the following Ada example:

```
1  PROCEDURE P IS
2  BEGIN
3     LOOP
4        IF X > 0 THEN Y := 2;
5        IF Y < 0 THEN Z := 3;
6     END LOOP;
7  END P;
```

The error token is "LOOP" on line six. During the scope recovery trial taking place at the preceding "END," the insertion prior to "END" of the closer token sequence "END IF;" is checked. This check does not parse into the right context, but does advance through the candidate itself. When the scope recovery routine is recursively invoked, the candidate sequence is eventually extended by "END

IF;," resulting in the candidate "END IF; END IF;." (In Section 3.1.2 we discuss how, for efficiency, the openers on the parse stack and in the deferred token sequence—in this case "LOOP" and the two "IF" tokens—are used to guide the generation of closer token candidates.) This candidate checks successfully to the end of the procedure, resulting in the following recovery:

```
1  PROCEDURE P IS
2  BEGIN
3    LOOP
4      IF X > 0 THEN Y := 2;
5      IF Y < 0 THEN Z := 3;
                              ↑
*** Syntax Error: "END IF;" inserted to match "IF"
                              ↑
*** Syntax Error: "END IF;" inserted to match "IF"
6    END LOOP;
7  END P;
```

2.4 Secondary Recovery

If scope recovery also fails, secondary recovery is invoked. The secondary recovery mechanism resumes the parse by discarding a token sequence immediately preceding the error token and/or a token sequence that begins with the error token and extends zero or more tokens into the right context. This mechanism can also involve the closing of open scopes.

Those trial tokens that were in the deferred token queue prior to entering error recovery could now be returned to it for secondary recovery. Our experience, however, is that one level of deferral (i.e., deferral of the actions induced by the error token) enhances the quality of secondary recovery, but greater deferral does not. In preparation for secondary recovery, then, the parse is advanced to the left-context point of the error token, with the deferred token queue remaining empty. Recall (Section 2.2.1) that prior to simple recovery, the parse configuration is set to the left-context point of the leftmost trial token (with the deferred token queue emptied), and remains there throughout simple and scope recovery (until a simple or scope repair is made). Now the trial tokens (except for the error token) are removed from TOKENS, and their corresponding actions applied to the state and parse stacks, advancing the parse to the left-context point of the error token (with the deferred token queue remaining empty). The state stack may alternatively be updated by simply setting it to the configuration held by STATE_STACK_COPY during the last trial of scope recovery.

Starting with the error token as the current token, it is checked whether parsing can resume by simply discarding a portion of the state stack along with possibly inserting one or more scope closer sequences. Thus a move down the state stack takes place, where it is checked at each point whether the parse can be resumed by cutting back the state stack this far. If the parse can be resumed at any point, recovery consists of peeling back the state and parse stacks to this extent. This iteration is performed all the way down the state stack. In our implementation the length of the parse check performed to test whether parsing can resume is MIN_ADVANCE + 2 (MIN_ADVANCE + 3 when the current token is an identifier). If the parse cannot be resumed at the current token, it is deleted from TOKENS (the succeeding token becoming the new current token), and the iteration down the state 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.

Consider Pascal Example P.3, also drawn from the sample in [16]:

```
(P.3)
1  PROGRAM P(INPUT, OUTPUT);
2  BEGIN;
3    PROCEDURE FACTR(N: INTEGER; VAR FACTOR: INTEGER);
4    BEGIN
5      X := 1
6    END;
7    X := 1
8  END.
```

The "BEGIN" in line 2 is misplaced: It should occur after the procedure declaration, just before the assignment to X on line 7. The desirable repair here is to "move" the "BEGIN" to its proper position. The semicolon succeeding this "BEGIN" complicates matters: Without it, a simple recovery would first result in the deletion of "BEGIN" in line 2, and an ensuing simple recovery would insert "BEGIN" after the semicolon of line 6. But, in the presence of this semicolon, the deletion of "BEGIN" immediately blocks. The deletion of this semicolon along with the "BEGIN" (followed by the insertion of "BEGIN" as described above) is an appropriate recovery but requires a multiple token deletion. We now trace the action of our error recovery routine on P.3. The error is detected at "PROCEDURE." The only simple repair candidates to parse into the right context are generated during the trial taking place at "PROCEDURE": The deletion of "PROCEDURE," as well as the substitution for it by semicolon, "BEGIN," "REPEAT," "CASE," "IF," or "WHILE," checks three tokens into the right context, blocking at the first colon. These candidates all involve the deletion of a keyword and are eliminated by our pruning criteria (see Section 3.2.1). Scope recovery also fails. Secondary recovery is then invoked, starting at the error token ("PROCEDURE"). The parse stack is set to the configuration obtained after the shifting of the semicolon:

[*program prog_head decl_part* "BEGIN" *stmt_list statement* ";"]

Note that both the *stmt_list* and *statement* are empty and so span no tokens. The deletion of the suffix of the parse stack starting at "BEGIN" in effect deletes "BEGIN;" and checks successfully until "X" in line 7. This secondary recovery is effected, and an ensuing simple recovery inserts "BEGIN" appropriately:

```
(P.3)
1  PROGRAM P(INPUT, OUTPUT);
2  BEGIN;
   ←-----→
*** Syntax Error: Unexpected input
3    PROCEDURE FACTR(N: INTEGER; VAR FACTOR: INTEGER);
4    BEGIN
5      X := 1
6    END ;
          ↑
*** Syntax Error: "BEGIN" expected after this token
7    X := 1
8  END.
```

3. IMPLEMENTATION CONSIDERATIONS

3.1 Performance Problems

The practicality of our method, especially the general version, depends on techniques that limit the size of the search space for simple and scope repairs. We accomplish this by limiting the number of repair trials and the number of closer token sequences considered in scope recovery. In a context where space constraints are severe or time performance is a high priority, we offer versions of the method that impose restrictions on the general model, such as the deferred version described in Section 2.2.1. Versions involving more limiting restrictions are described in Section 3.1.3.

3.1.1 *The Number of Repair Trials.* We mentioned in Section 2.2.1 that a special technique is used to determine the necessary number of simple and scope repair trials. A simple or scope repair is not feasible to the left of a parse stack element or a deferred token if it is not possible in any program context to parse from that point through the error token. Thus a technique for recognizing when a sequence of symbols cannot appear within any sentential form of the language can be used to limit the number of repair trials. We have developed techniques to efficiently accomplish this within both the LR and LL frameworks: See Section 4.3 for measurements of their effectiveness in limiting the number of repair trials.

We now describe our LR technique. Suppose the sequence W of symbols under consideration is $[W_1W_2 \ldots W_k]$, where each symbol W_i may be a terminal or a nonterminal. With respect to determining the possible legality of the above sequence, a nonterminal may be regarded as representing a choice of any of the terminal strings that it derives. For symbol W_1, SHIFT_STATES (W_1) consists of the set S of states that can be entered as a result of shifting W_1. If for each state s in S it is not possible to parse the token sequence $[W_2 \ldots W_k]$ when starting with s as the top element of the state stack, then the sequence W cannot legally appear.

A special parse procedure PARSE_BLOCK is used to determine if the parse will surely block when the state s is on top of the state stack and a given token sequence is a prefix of the remaining input tokens. Of course it is not always possible to determine whether the parse must block: A reduction may involve cutting back the state stack below s. But in this case, if the current token is a terminal and does not belong to the FOLLOW set of the rule being reduced, we can conclude that the parse must block. The technique employed here of "starting up" the parse at an arbitrary token (via SHIFT_STATES) is borrowed from the "forward move" algorithms of [12] and [13] that restart the parse to condense the right context of an error.

The number of repair trials is determined by applying this legality check to sequences of symbols that immediately precede and include the error token ET. Where the deferred tokens configuration is $[DT_1DT_2 \ldots DT_k]$, the check is applied to the sequence of tokens from DT_i through ET, where i is first set to k and then decremented by one down to one. If for some i the sequence $[DT_iDT_{(i+1)} \ldots DT_kET]$ fails the legality check, then it is unnecessary to perform a trial at

any of the tokens preceding DT_i or any of the parse stack elements (a trial would be held at each element of this sequence). If the sequence $[DT_1DT_2 \ldots DT_kET]$ passes the legality check, then, in the general version, trials may also be held at parse stack elements. Suppose the parse stack configuration is $[P_1P_2 \ldots P_n]$. The legality check would be applied to sequences of symbols of the form $[P_jP_{(j+1)} \ldots P_nDT_1DT_2 \ldots DT_kET]$, where j is first set to n and then decremented by one down to one. If a sequence proves to be illegal, trials are not held below that point on the parse stack (or at that point, if P_j is a nonterminal, as nonterminals are not to be deleted or substituted for).

In the LL framework, an analogous technique is used. Here the set SYMBOL_RULES(W1) of rules whose right-hand sides contain W1 is considered. If for each such rule the portion of its right-hand side to the right of W1 does not successfully predict the input sequence $[W_2 \ldots W_k]$, then the sequence $[W_1W_2 \ldots W_k]$ cannot legally appear. Here the procedure PARSE_BLOCK determines whether the parse must block when a given sequence of symbols is on top of the prediction stack and a given token sequence is a prefix of the remaining input tokens. Once again, FOLLOW sets are used. See [3] for a more detailed description of the LL determination of the number of repair trials.

3.1.2 *Making Scope Recovery Efficient.* Scope recovery involves closing open scopes: An inserted closer token sequence always matches some scope opener that has been seen and not yet closed. For example, a right parenthesis matches a left parenthesis, and in Ada the sequence "END IF;" matches the opener "IF." Other examples of openers in Ada and Pascal are "PROCEDURE" and "BEGIN." A natural approach toward scope recovery, then, is to examine the parse stack and the deferred token sequence for unclosed openers, and let these determine the closer token sequences that are considered for insertion. For efficiency, we opted for this approach in our implementation (for more detail, see Section 2.3 of [3]). At the point of error detection in the example of Section 2.3, the parse stack contains the openers "LOOP," "IF," and "IF." In that particular recovery, the presence of the topmost "IF" induces the generation of the candidate "END IF;"; the presence of the other "IF" induces the extension of this candidate to "END IF; END IF;." Note that two additional language-dependent maps are required by this approach: OPENERS and CLOSER_MAP. For each opener in OPENERS, for example, "IF," CLOSER_MAP gives the set of associated closer token sequences in CLOSERS, for example, "END IF;."

3.1.3 *More Efficient Versions of the Method.* The generation of left-context repair candidates has two dimensions: the consideration of parse stack elements and the deferring of parse actions. Given these two dimensions, there are four recovery models to choose from. In Section 2 we described the general version, which incorporates both dimensions, and the deferred version, which does not consider parse stack elements.

The *minimal version* does not defer parse actions or consider parse stack elements, and so only performs a single trial at the error token. It may not even consider an appropriate repair at the error token when premature reductions occur. In addition to speeding up error recovery, it incurs no space or time overhead with respect to the parsing of correct programs.

The *condensed version* does not defer parse actions, but considers parse stack elements in simple and scope recovery. An error in the prefix might be repaired, but there is still a reliance on the appropriateness of reductions. Premature reductions, particularly default reductions in the LR framework, are still troublesome here.

In some contexts the choice of a version that does not perform trials at parse stack elements (i.e., the minimal or deferred version) is advantageous. Performing trials at parse stack elements in simple and scope recovery requires that the parser (or at least the parse checker) have the capability of parsing nonterminals in contexts in which they are not necessarily legally expected. This imposes a space cost when the LR parse table scheme saves space by assuming that goto actions on nonterminals do not need to be checked for legality [10]. When parsing a correct program, this assumption is always valid. But suppose that an error has been encountered, and an insertion is attempted at a point below one or more nonterminals on the parse stack. The parse check routine must determine whether these nonterminals are expected in this altered configuration, and so it cannot be assumed that they are legal. The parse table scheme must then allow for checking nonterminal gotos, resulting in larger tables. The deferred version does not perform trials at parse stack elements and so is compatible with this space-efficiency technique. (A space-efficient LR implementation of the general and deferred versions of our method is discussed in Section 4.2.2.)

In the LL framework, accommodating nonterminals as possible input symbols poses difficulties even where the symbol can be presumed legal (see [3]). When a nonterminal A is the next input symbol and is also predicted, it can be shifted—that is, deleted from the prediction stack and pushed onto the parse stack—as if it were a terminal. The more difficult case is the one in which a different nonterminal B is predicted. This is only a legal configuration if there exists a sequence of rules such that B derives a string of symbols beginning with A. A function FIND_RULE is needed that returns the first rule in this sequence if it exists. Execution of the function involves much more than a single table lookup, and in fact it invokes the LL parse action lookup routine twice for each terminal symbol in the grammar.

Our method makes no assumptions about the timing of semantic actions. Semantic analysis may take place in a separate pass from syntactic analysis, or it may accompany parsing. If semantic actions are performed during parsing only as reductions occur, then a manipulation of the terminal symbols on the parse stack (as occurs during simple and scope recovery) cannot invalidate semantic information that has already been generated. But suppose that semantic actions occur at other points also, such as "new scope" semantic actions that might be performed as a "BEGIN" symbol is shifted. Then any manipulation of the parse stack may render semantic information invalid, and so backing down the parse stack potentially interferes with semantic processing. In such a front end, the deferred version offers the advantage of effecting simple and scope recoveries without necessitating the disabling of semantic processing. In a syntax-directed translation scheme that does more than build an abstract-syntax tree, however, it may be necessary to turn off semantic actions when secondary recovery deletes one or more nonterminal symbols.

3.2 Heuristic Evaluation of Repairs

3.2.1 *Pruning of Repair Candidates.* The evaluation of simple repair candidates, as described in Section 2.2.3, may begin with the elimination of some candidates through the application of heuristic criteria, including language-dependent preferences. Our experience with Pascal and Ada samples has suggested that certain criteria are generally effective in guiding the choice of a simple repair candidate. Although their plausibility should be evident, these should not be construed as hard and fast rules.

One general observation is that repairs involving the insertion and/or deletion of keywords are dangerous in that spurious such repairs often parse check beyond the minimum required distance and can significantly alter the meaning of the program text. Thus we apply stricter criteria to keyword repairs, possibly pruning some or all of them. One rule is that, if a given repair mode has both keyword and nonkeyword candidates, the keyword candidates are eliminated. Recall (Section 2.2.3) that three simple repair candidates parse check equally well in Example P.2: deletion of the error token "ELSE," deletion of the semicolon, and substitution of a semicolon for "ELSE." In pruning candidates that delete keywords when candidates that delete nonkeywords are also present, the deletion of "ELSE" is eliminated as a candidate. Deletion is favored over substitution as a repair mode, and so deletion of the semicolon (the most appropriate repair) is favored over the remaining substitution candidate.

We also prune keyword repairs that do not meet a long parse check criterion (MIN_ADVANCE + 3 in our implementation). Recall (Section 2.4) Pascal fragment P.3:

(P.3)

```
1  PROGRAM P(INPUT, OUTPUT);
2  BEGIN;
3     PROCEDURE FACTR(N: INTEGER; VAR FACTOR: INTEGER);
4     BEGIN
5        X := 1
6     END;
7     X := 1
8  END.
```

We noted that the deletion of "PROCEDURE" and several substitutions for it are simple repair candidates that parse check three tokens into the right context (up until the colon). They all would result in poor recoveries, and in fact they do not meet our long parse check requirement for keyword repairs and so are pruned. This example illustrates that inserting or deleting a keyword that opens a scope is especially dangerous.

In addition to pruning candidates in accordance with conservative criteria toward keyword repairs, we also allow for the heuristic evaluation of repair candidates as guided by the language-dependent sets ALWAYS_PREFERRED and PREFERRED_FOR. Where the implementor chooses to make use of these sets, they specify preferred insertion and substitution candidates. See Appendix A for the assignments to these sets in our Pascal and Ada implementations. Where one or more insertion candidates are preferred (i.e., belong to the set ALWAYS_PREFERRED), then the other insertion candidates are

eliminated. Similarly, if a substitution of symbol y for x is a candidate and $[x, y]$ is an entry in the language-dependent set PREFERRED_FOR, then all other substitutions (except those similarly preferred) are eliminated. These preferences are given precedence over the rules for eliminating keyword repairs: That is, a preferred insertion or substitution is never eliminated on the basis of involving a keyword.

3.2.2 *Preference Order Among Repair Modes.* Recall the preference order among repair modes stated in Section 2.2.3: merge, misspelling, insertion, deletion, substitution. Top preference is given to merge and misspelling because it is unlikely, where such a repair checks, that is not the repair of choice. It is natural to give substitution the lowest preference, since it may be thought of as an insertion combined with a deletion. We illustrate the effectiveness of the preference for insertions over deletions and substitutions with the following erroneous statement from the sample in [16]:

IF $N \leq$ THEN *POWER* := ELSE *POWER* := M * *POWER*(M, $M - 1$)

Our implementation inserts an identifier after "≤" and also after ":=." In that expressions seem to have been intended at these points, these are excellent recoveries. With substitution given preference over insertion, "↑" (the dereference operator) is substituted for "≤," resulting in an unhelpful diagnostic. With deletion given preference over insertion, ":=" is (inappropriately) deleted. For several other examples in the sample in [16], giving substitution preference over insertion results in a poorer recovery. Otherwise, permuting the preference order of these three modes seldom affects the quality of a recovery; it affects none in the sample in [16]. It is always possible, of course, that permuting our recommended order may enhance the quality of a particular recovery.

3.2.3 *Controlling Secondary Recovery.* There are two dimensions to a secondary recovery: that of peeling back the state and parse stacks (eliminating left context), and that of advancing past the error token and succeeding tokens (eliminating right context). As described in Section 2.4, our method is biased toward preserving right context and discarding left context. This does not always result in a secondary recovery that deletes the minimum amount of overall context, or that is the most appropriate secondary recovery by any other criterion. We experimented with techniques for eliminating this bias, including the development of a formula to assign a certain cost for deleting a stack entry, and another cost for deleting the current or an advance token. Such a formula would provide a basis for choosing one secondary recovery over another, rather than simply opting for the first one to parse check the required distance. But we found that our simple approach fares just as well in general as a more complicated strategy that would occasionally preserve more left context. The basic reason for this is that rarely in practice are more than two state stack items removed, and so generally not much left context is discarded. We did find, however, that, where a secondary recovery discards no left context and only a small amount of right context, it is usually the most appropriate action. For example, consider the following erroneous Ada procedure header:

PROCEDURE Q(Z: RANGE 1 .. 10; X: INTEGER) IS

The type specification for the parameter Z is erroneous: A typemark should be supplied, and in this context a range constraint is not allowed [2]. The error token is "RANGE": The deletion of it and right context up until "INTEGER" is a good recovery. But secondary recovery as described in Section 2.4 would delete the sequence of tokens beginning at the left parenthesis and running through the "10," since "PROCEDURE Q" in the left context is consistent with "; X: INTEGER" in the right context. The procedure declaration of Q has then been taken to be a procedure specification, resulting in a poor recovery.

Thus in our implementation we begin secondary recovery with a preliminary phase that determines whether the parse can be resumed by deleting the error token along with some prefix of the token sequence comprising the right context. The deletion of two, three, up to some bounded number of tokens of right context is tried, and the first such multiple deletion that allows the parse to advance a distance of MIN_ADVANCE + 2 tokens into the remaining right context is taken. The deletion of right context in this manner is bounded by the occurrence of one of a language-dependent set BEACONS of symbols that are not to be deleted in this mode of recovery. Note that the symbols in BEACONS do not play the role that beacon symbols do in traditional panic mode recovery [9].

3.3 Diagnostics

Careful attention has been paid to 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 Appendix B 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) or the predict stack (LL), then the deleted input may be viewed as a malformed statement, and so the diagnostic message "Bad statement" is issued. When no such message is available, the bad input is simply termed "Unexpected." We specify an association between the list nonterminal symbols and their corresponding messages in the parser generator, which then generates a map that allows the parser to associate a state with its apropriate message. This process could be further automated by having the parser generator deduce the messages automatically from nonterminal textual names.

4. MEASUREMENTS AND EVALUATION OF PERFORMANCE

In Section 4.1 we present empirical measurements of the time performance of implementations of the method on sample erroneous Pascal and Ada programs. We have stated as a goal (Section 1.1) that in practice the time spent in repairing a single error be independent of the length of the program. This requirement is in fact satisfied: The average times given in Sections 4.1.1 (for Pascal) and 4.1.2 (for Ada) hold for programs of any length.

In our implementations the extent of the forward parse check is not actually the entire program text in the right context of the error but is bounded in length

by a constant MAX_LOOK_AHEAD that we set to 25. MAX_LOOK_AHEAD should be large enough to accommodate the possibility that the erroneousness of the repair candidate is detected late. The length of 25 is generally sufficient, but in the case of a scope correction or the insertion of a candidate that opens a new scope, a longer check may sometimes help. Otherwise we have not encountered an example in which a check longer than 15 tokens in advance of the error token is needed.

The other time-efficiency factor for the general version is the size of the parse stack, in that it limits the number of trials that occur. In the presence of non-left recursive grammatical structures, the parse stack may grow with the length of the program, but in general the average parse stack size does not vary from shorter to longer programs. In any case it is not the average parse stack size that is directly relevant, but rather the average number of trials. We have found that the average number of trials does not vary with program length (see Section 4.3).

4.1 Measurements

The LR parsers that we have implemented and experimented with are LALR. Default reductions are applied in any state that has only one completed rule (see [3] for an examination of the trade-offs involved in applying default reductions to different extents). All versions of the parser have been implemented as separate parse modules attachable to a translator writer system, all written in the very high-level language SETL. The application of our system to a particular language amounts to writing the lexical module for that language, since all language-dependent features are included there. This module essentially consists of a lexical analyzer and a procedure that sets the error recovery parameters. We have written and tuned such modules for Pascal and Ada.

4.1.1 *Pascal.* We ran all versions of the parser on the sample in [16], supplied by Ripley and Druseikis (obtained by them from the original sample in [17] by reducing it to "unique" syntax errors). We refer to this sample as "PTESTS." Each version has also been run on a program (PTESTS1) that results from removing all errors from this sample. Subtracting the time spent by a particular version parsing PTESTS1 from the time it spends parsing PTESTS and dividing by the number of errors give the average time per error for this version. The programs were executed on a VAX 11/780. Our experience with recoding SETL programs into a lower level language, such as PL/1, suggests that a speedup factor of 30 can be achieved through a simple transliteration of the SETL codes [6].

The quality of a recovery is measured using categories proposed by Pennello and DeRemer [13], whereby a repair is rated "excellent" if it is the one a human reader would make, "good" if it results in a reasonable program and no spurious or missed errors, and "poor" if it results in one or more such errors or if excessive token deletion occurs. At times two different repairs of an error reflect syntactically equivalent views of what the error is. For example, in a case where two consecutive commas occur among a sequence of identifiers that legally are to be separated by single commas, the deletion of a comma is essentially syntactically equivalent to the insertion of an identifier. We have used [16], which provides an interpretation of the syntax error involved for each case in the sample, to

guide our determination of the repair that the hypothetical human reader would effect. Results obtained with the sample in [16], for both the LR and LL general versions, with the extent of deferral k set to one are as follows:

Excellent	Good	Poor
128	33	4
(77.6%)	(20.0%)	(2.4%)

Pennello and DeRemer [14] rate their own recoveries on the sample in [16] as follows:

Excellent	Good	Poor
64.9%	29.4%	5.7%

With the scope recovery phase, which relies essentially on language-dependent sets, turned off, eight recoveries are affected. Our recoveries then rate as follows:

Excellent	Good	Poor
121	36	8
(73.3%)	(21.8%)	(4.8%)

The four additional poor recoveries without scope recovery are excellent recoveries with it. They are all missing the END statement, or the entire statement part, of a procedure or program; the poor recovery is to delete the entire procedure or program.

Increasing k from one to two in the general versions results in the improvement to Example P.2 as discussed in Section 2.1.3, where a semicolon precedes "ELSE." This error may arise owing to confusion regarding the role of the semicolon as a separator rather than as a terminator and so is probably a common mistake among beginning Pascal programmers. But, as the only recovery in the entire PTESTS sample that is affected in a significant way by the extension to two levels of deferral, it is probably insufficient to justify the extension. Further increasing k does not improve the performance of the general version. Thus, with respect to Pascal, the appropriate degree of deferral for the general version seems to be one level.

The deferred versions perform as well as the general ones on this sample, so long as the extent of deferral k is at least 12 (see Example P.1 of Section 2.1.3). With k set to one, five recoveries (all in cases essentially identical to P.1) rated as excellent for the general version worsen to good in the deferred version.

The time spent per error by the general and deferred versions of our method is about 4.50 seconds (and so about 0.15 seconds, given the speedup factor). With the more economic versions of our method, the time per error is about half this, and performance still rates comparably with Pennello and DeRemer (see [3]). With k set to one, LR and LL provide recoveries of equal quality in all examples in the sample: The few differences between them are minor.

4.1.2 *Ada.* One particular Ada test program that we developed, ATESTS (see the section on Ada in Appendix B), is about 75 lines in length and includes 36 errors that we regard as either common or troublesome. ATESTS1 is the same Ada program but with these errors removed. The LR results with this sample are summarized in Table I.

Table I. LR Results with ATESTS

Version	K	Number of repairs rated as: Excellent	Good	Poor	ATESTS time	ATESTS1 time	Average time per error
General	2	33	3	0	5:23	36	7.97
General	1	30	4	2	4:50	36	7.06
Deferred	3	33	2	1	5:32	36	8.22
Deferred	2	31	2	3	4:14	36	6.06

This short sample is sufficient to demonstrate that the extent to which reductions are deferred more seriously influences recovery actions on Ada programs than on Pascal programs. In three cases the performance of the general version is improved by setting k to two instead of one. All three cases involve misspelled reserved words, and in all three cases, the error is detected one token late: That is, the erroneous (misspelled) token is PREVTOK at the time of error detection. Given a single level of deferral, PREVTOK is guaranteed to be the top symbol on the parse stack at the point of error detection, and so substitutions for it are generated as candidates. But in each case the erroneous identifier, prior to being shifted, induces a reduction by a rule with an empty right-hand side that disallows the misspell substitution from successfully parse checking.

Two cases in which an inferior recovery takes place with only a single level of deferral involve the introduction of an empty statement label list onto the parse stack. Consider the following Ada segment:

```
CASE M IS
  WHEN FEB ⇒ RETURN 28;
  WHAN APR ⇒ RETURN 30;
```

The following rules of the Ada grammar [2] are relevant here:

```
case_statement ::= CASE expression IS
                   pragma_list
                   case_statement_alternative
                   case_statement_alternative_list
                   END CASE;
case_statement_alternative ::= WHEN
                               choice choice_list ⇒
                               sequence_of_statements
sequence_of_statements ::= pragma_list statement statement_list
```

Also, a statement begins with a (possibly empty) label list. After the first semicolon has been shifted, the parse stack contains the suffix

[CASE *expression* IS *pragma_list* WHEN *choice choice_list*
"⇒" RETURN *expression* ";"].

Where $k = 2$, this is the parse stack configuration at the point of error detection, and the identifier "WHAN" is the only element of the deferred tokens queue. But with $k = 1$, this identifier is presumed to begin a second statement of the sequence of statements that may follow the "⇒" (note that it is not regarded as possibly being a label, since a label must begin with the symbol "<<"). Thus it induces the reduction of [RETURN *expression* ";"] to a sequence of statements

and then the pushing of the nonterminal *label_list* onto the parse stack as an empty label list. The above parse stack suffix has become

[CASE *expression* IS *pragma_list* WHEN
choice choice_list "⇒"
sequence_of_statements label_list IDENTIFIER].

Owing to the presence of the (empty) label list, the substitution of "WHEN" for "WHAN" does not successfully parse check.

The third case, in which the reserved word "separate" is misspelled, involves the introduction of an empty *basic_declarative_item_list* onto the parse stack. This case is discussed in detail in [3]. All three inferior recoveries result from a premature introduction of a list by means of a reduction by an empty rule, evidence of the troublesome nature of these reductions for error recovery.

An alternative solution to deferring parse actions an extra level is to rewrite the syntactic rules defining lists so that they cannot be generated by an empty rule. The general technique is to replace a list of zero or more items by an optional list of one or more items. A drawback of this approach is that it enlarges the size of the grammar and so of the parse action table. See Section 4.2.3 of [3] for a full description of this technique and measurements of its cost with respect to table size.

The implementor's choice of a grammar, however, may be governed by other concerns, such as compatibility with a standard. For example, the implementor may choose to follow the rules of the Ada standard [2], whether it is an ideal grammatical design or not. In any case deferring parse actions an appropriate extent is preferable to grammatical tuning in that the choice of an error recovery should not be dictated by grammatical design.

Another difference that we have observed between Ada and Pascal is that scope recovery is more frequently of importance in handling syntactic errors in Ada programs. Several of the examples in our Ada sample (such as the example of Section 2.3) were chosen by us because we felt the omission of a sequence of closers to be a common error, in particular for scope constructs that do not require closers (or do not exist at all) in other languages. The syntactic complexity of Ada, along with its abundance of scope constructs, makes for larger scope recovery sets for it (see the section on Ada in Appendix A).

The general version of our method (with only a single level of deferral) is currently in use in the NYUADA/ED translator and interpreter, and so has been tested on the Ada Compiler Validation (ACV) test suite. We have not measured our performance on the numerous syntactic error tests among these since there is no reason they should be taken to be representative of the kinds of errors that any group of programmers (beinning or expert) would tend to make, and certain types of errors occur repeatedly. We have in general done well on this sample, and some of the examples in ATESTS have been drawn from it.

4.2 Overhead to Correct Programs

4.2.1 *Time Overhead of Deferral Mechanisms.* The deferring of parse actions to some extent $k > 0$ inevitably adds to the time cost of parsing a correct program. We represented the deferred action queues with SETL tuples, using costly

high-level tuple operations (such as concatenation) without availing ourselves of the SETL representation specification facility that allows the programmer to indicate the type of the elements of the tuple and/or a data structure to efficiently implement it.

In both our LR and LL implementations, the time cost of deferring parse actions is less than 10 percent. This percentage increase is with respect to the time cost of lexical analysis and parsing (without semantic analysis), which generally accounts for 20–40 percent of the time spent compiling a program.

It is difficult to imagine a context in which the above cost would be prohibitive, but it is worth noting that even the most general version of our method can be implemented so that no time overhead at all is incurred for correct programs. We have already observed that parse deferral may be viewed as double parsing. One can accomplish this "double parse" with only a single parse for correct programs by letting the first parse proceed until an error is encountered. At that point the second parse is invoked and proceeds the desired distance k from the error token. The effect of a deferral of k is then accomplished, with the advantage that, where no syntactic error is detected, no second parse takes place. Where the programmer expects that he or she has a correct program, he or she may wish to parse it in this "no time overhead" mode.

4.2.2 *Space Overhead.* Except for the general LL version, for which the FOLLOW map is required by the FIND_RULE routine (Section 3.1.3), the only space overhead derives from the determination of the number of repair trials by the parse block check described in Section 3.1.1. The inclusion of this technique is desirable from the viewpoint of time efficiency. The parse block routine requires FOLLOW and SHIFT_STATES maps in the LR framework, and FOLLOW and SYMBOL_RULES maps in the LL framework. The maximum requirements for additional maps are summarized in Table II.

In some LR parsers, the general (but not the deferred) version imposes a space overhead in requiring the capability of parsing nonterminals in contexts in which they are not necessarily expected (Section 3.1.3).

Where space efficiency is a higher priority than time efficiency, the number of repair trials may simply be set to a fixed number. Space overhead is then entirely eliminated in the LR context (the FOLLOW map would still be necessary for the general LL version).

The minimal LR and LL versions impose no space overhead.

4.3 The Number of Repair Trials

The effectiveness of the "parse block" check described in Section 3.1.1 in limiting the number of repair trials is a critical efficiency concern for the general version and for a deferred version with a long deferral. Tables III and IV indicate the effectiveness of this technique for the general version. They indicate the trial distribution for all recoveries, including those where simple and scope recovery fail. Note that the average number of trials grows with the degree to which parse actions are deferred. For $j \leq 10$, the jth entry of each row indicates the number of recoveries in the sample for which j trials took place.

Table II. Maximum Space Overhead

Version	FOLLOW	SHIFT_STATES	SYMBOL_RULES	Percent increase in table sizes	
				Ada	Pascal
LR	Yes	Yes	No	10.7	10.8
LL	Yes	No	Yes	—	11.1

Table III. Trial Distribution for All Recoveries for PTESTS

Version	K	Trial distribution										Average number of trials
		1	2	3	4	5	6	7	8	9	10	>10	—
LR	1	0	115	35	5	15	21	0	2	0	0	0	2.96
LR	2	0	83	38	8	22	19	8	7	1	4	3	3.81
LR	3	0	83	31	6	14	8	17	18	4	8	4	4.29
LL	1	0	102	38	6	12	18	9	2	2	0	2	3.32
LL	2	0	82	39	6	18	24	7	3	6	3	3	3.84
LL	3	0	82	32	3	9	22	23	5	6	5	4	4.20

Table IV. Trial Distribution for All Recoveries for ATESTS

Version	K	Trial distribution										Average number of trials
		1	2	3	4	5	6	7	8	9	10	—
LR	1	1	22	4	4	0	1	1	0	0	1	2.82
LR	2	1	19	4	3	1	1	0	1	2	2	3.53
LR	3	1	19	3	2	2	1	0	2	1	3	3.73

5. CONCLUSION

5.1 Summary of Results

On the basis of the literature that we have surveyed, the method of Pennello and DeRemer seems a legitimate representative of the current state of the art in syntactic error recovery. Fortunately, they provide a measure of the effectiveness of their method on the same Pascal sample as ours. A comparison of their performance statistics with ours (Section 4.1.1) provides evidence that we have in fact succeeded in developing a method that advances the state of the art in terms of effectiveness, and even our limited "efficiency" versions perform comparably with theirs.

The measurements of Sections 4.1.1 and 4.1.2 also indicate that the average time per repair of even the most general version of our method is reasonably small. And in Section 4.2 we have seen that the space and time overheads incurred by our most general version amount to slightly more than 10 percent and less than 10 percent, respectively. These costs would not seem to be

prohibitive with respect to the speed requirements or spatial constraints of almost any application. Time overhead for correct programs may in any case be eliminated entirely by effecting parse deferral in the manner described in Section 4.2.1.

Most of the literature on syntactic error recovery confines its empirical studies to Pascal programs. But, owing to Ada's higher syntactic complexity, syntax errors tend to pose more of a difficulty in Ada than Pascal programs. In that we have applied the method with success to both Ada and Pascal, there is some empirical evidence for our claim that the method is essentially language independent. We have found that with Ada the late detection of an error is more likely to occur. Our parse action deferral mechanism is low cost and makes it possible to handle difficult examples of errors that are detected late, as we have seen with several of the examples drawn from ATESTS. We have also found that, with Ada, language-dependent parameters play a more important role with respect to the quality of our recoveries.

A significant result is that our general LR and LL versions perform equally well on all PTESTS examples. The method thus shows itself to be equally applicable to LR and LL parsing. This result suggests that there is not much to choose between LR and LL as far as the quality of error recovery is concerned.

5.2 Comparison with Other Methods

Our method builds upon our own earlier work [4], as well as that of Feyock and Lazurus [7], Graham, Haley, and Joy [9], and Poonen [15]. The recovery methods of [7] and [9] 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 on identification of an ill-formed program component. Both methods test single token repair candidates by performing a separate forward parse check for each, but they depart from ours in using semantic information when evaluating a candidate. We have found that our repair selection criteria and language-dependent sets (Section 3.2) preclude the need for the weighted cost analysis of [9]. The techniques that we use in generating the set of simple repair candidates, including the parse action deferral mechanism (Section 2.1.3) and the determination of the necessary number of simple recovery trials (Section 3.1.1), are original.

The mechanism in [7] for backing up the parse bears a resemblance to our parse action deferral mechanism, but rather than deferring the parse actions for recently parsed tokens, it simply places these tokens in a buffer. In that actions already applied to the parse stack are undone as the parse is backed up, the integration of the semantic and parsing phases of the compiler then seems problematic. And their reliance on semantic information for analyzing repair candidates indicates the presumption of a parse model in which these phases are integrated. Their backup halts at the beginning of what is taken to be the relevant program component (the "substructure"). It is not checked whether this far a backup is necessary. If the substructure contains more than one error, it is deleted: Unlike us, they make no effort to repair errors in close proximity.

Poonen and Johnson [10] base their recovery algorithms on a stack resynchronization technique that is similar in spirit to the second phase of our secondary recovery. But their methods depend on the augmentation of the grammar by

error productions. Poonen bases resynchronization with right context on a single occurrence of any member of a set of tokens, rather than on a sequence of advance tokens. We do not use error productions. They have the advantage of speeding up secondary recovery, but complicate the grammar and in most cases provide a diagnostic that could just as well be derived automatically from the parse or prediction stack. Graham, Haley, and Joy [9] 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 preliminary forward move of [12] and [13] 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 parser generator [14].[2] We find that our implementations spend little time parse checking in simple recovery, although the check is allowed to be long if necessary and may be repeated for many candidates. Typically, in the course of a simple recovery few candidates require a check of more than two tokens.

The preliminary forward move lacks a systematic method for dealing with the presence of errors in the right context. Mauney and Fischer address this issue in [11], describing an algorithm that repairs all errors within a "region" that includes the error token and some right context. They adapt Aho and Peterson's algorithm [1] for finding the least-cost repair of an entire program, applying their adaptation to the region. This approach is expensive with respect to time, however, unless the region is very small, and, in the LR framework, requires a considerable augmentation of the original grammar. With our method an error in the right context may sometimes prevent a simple recovery from taking place, since a plurality of candidates may check up to the occurrence of the second error, but not so far as the required threshold (see Section 2.2.2). However, we have good success at handling errors in close proximity (see examples in Appendix B). Without a systematic method for repairing multiple errors, multiple symbol deletions are of particular importance, and our secondary recovery is designed especially to handle these cases. In [12] and [13], a multiple symbol deletion involving a mutilated right 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 accommodated 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 repair 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 repair.

We have found that our scope recovery mechanism, at little cost, significantly enhances many of our Ada recoveries (see the section on Ada in Appendix B). We are unaware of the systematic incorporation of this form of recovery in any other method.

[2] Given a nondeterministic implementation of the forward move algorithm, the only additional tables required are the FOLLOW sets (private communication with F. L. DeRemer).

APPENDIX A. LANGUAGE-SPECIFIC SETS

Pascal

The language-specific sets for our Pascal implementations are as follows:

```
PREFERRED_FOR := {
  [':=', '='],      $ For := used instead of = in an expr
  ['=', ':='],      $ For = used instead of := in assignment
  [',', ';'],
  [';', ',']};
ALWAYS_PREFERRED := {'IDENTIFIER', ';', ',', '['};
BEACONS := {'PROGRAM', 'BEGIN', 'END', 'FUNCTION', 'PROCEDURE',
  'DO' 'FOR', 'REPEAT', 'UNTIL', 'WHILE', 'IF', 'THEN',
  'ELSE', 'CASE', 'WITH', ';', '.'};
OPENERS := {'prog.head', 'PROGRAM', 'PROCEDURE', 'FUNCTION',
  'ARRAY', '(', '[', 'REPEAT', 'BEGIN'};
CLOSERS := [
  ['BEGIN', 'END', '.'],
  ['END', '.'],
  ['BEGIN', 'END', ';'],
  [';'],
  [')'],
  [')', ';'],
  [']'],
  ['END'],
  ['UNTIL', 'IDENTIFIER'],
  ['UNTIL', 'IDENTIFIER', ';'],
  ['OF', 'IDENTIFIER']
      ];
```

Ada

The language-specific sets for Ada are as follows:

```
PREFERRED_FOR := {
  [':=', '='],      $ For := used instead of = in an expr
  ['=', ':='],      $ For = used instead of := in assignment
  ['.', '. .'],
  ['IS', ':='],
  [':=', 'IS'],
  ['IS', 'OF'],
  ['OF', 'IS'],
  ['identifier', 'PROCEDURE'],
  ['TYPE', 'SUBTYPE'],
  ['SUBTYPE', 'TYPE'],
  ['DO', 'LOOP'],
  ['FOR', 'LOOP'],
  [';', 'LOOP'],
  ['numeric_literal', 'identifier']
                                        $ For 1.100 instead of 1 . . 100 in index
                                        };
ALWAYS_PREFERRED := {'identifier', ';', ','};
BEACONS :=
  {'BEGIN', 'DO', 'ELSE', 'ELSIF', 'END', 'FOR', 'FUNCTION',
   'IF', 'LOOP', 'PACKAGE', 'PROCEDURE', 'TASK', 'THEN',
   'WHILE', ';', 'RETURN', 'IS'};
```

```
OPENERS :=
  {'subprogram_specification', 'TASK', 'PACKAGE',
   '(', 'IF', 'ARRAY',
   'SELECT', 'ELSIF', 'BEGIN',
   'CASE', 'LOOP', 'ELSE', 'RECORD'};
CLOSERS := [
  ['BEGIN', 'NULL', ';', 'END', ';'],
  ['END, ';'],
  [')'],
  ['END', 'IF', ';'],
  ['END', 'SELECT', ';'],
  ['END', 'CASE', ';'],
  ['END', 'LOOP', ';'],
  ['END', 'RECORD', ';'],
  ['IS'],
  ['IS', 'WHEN'],
  ['OF'],
  ['OF', 'identifier'],
  [')', ';']
    ];
```

APPENDIX B. SAMPLE RESULTS

Pascal

EXAMPLES FROM PASCAL SAMPLE PTESTS:
(General LR Version with level of deferral $k = 2$)
(Rating and type of each recovery is indicated immediately above it.)
(See [5] for listing of run with the entire sample PTESTS.)

Excellent Simple Recovery.

```
  1   PROGRAM P(INPUT, OUTPUT);
  2   BEGIN
  3     IF X := 0 THEN X := 1
              ↑
*** Syntax Error: "=" expected instead of ":="
      END.
```

Three Excellent Simple Recoveries.

```
  5   PROGRAM P(INPUT, OUTPUT);
  6     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"
  7     BEGIN
  8     END;
  9   BEGIN
 10    X := 1
 11  END.
```

Excellent Simple Recovery.

```
  12.  PROGRAM P(INPUT, OUTPUT);
  13.    VAR L, N: REAL;
  14.    VAR X, NONPRIME, PRIME: INTEGER;
             ↑
*** Syntax Error: Unexpected "VAR" ignored
  15  BEGIN
  16  END.
```

Six Excellent Simple Recoveries.

```
  17  PROGRAM P(INPUT, OUTPUT);
  18  BEGIN
  19    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 ","
  20 END.
```

Good Secondary Recovery.

```
  21  PROGRAM P(INPUT, OUTPUT);
  22  BEGIN
  23    FOR I := 1    STEP 1 UNTIL LISTSIZE - 1 DO
        <-----------------↑↑↑↑-------------------------------------->
*** Syntax Error: Bad statement
  24    X := 1
  25  END.
```

Excellent Simple Recovery.

```
  26  PROGRAM P(INPUT, OUTPUT);
  27  BEGIN
  28    FOR I := 1 TO MAXELEMENTS
                                  ↑
*** Syntax Error: "DO" expected after this token
  29    Y[I] := 0;
  30  END.
```

Excellent Simple Recovery.

```
  31  PROGRAM P(INPUT, OUTPUT);
  32  BEGIN
  33    FOR K1 := TO NOELEMS DO X := 1
              ↑
*** Syntax Error: IDENTIFIER expected after this token
  34  END.
```

Excellent Simple (misspelling) Recovery.

```
  35  PROGRAM P(INPUT, OUTPUT);
  36  BEGIN
  37    IF NON PUSH(I) THEN X := 1
            ↑
*** Syntax Error: Reserved word "NOT" misspelled
  38  END.
```

An Excellent Simple Recovery and an Excellent Scope (double insertion) Recovery.

```
  39  PROGRAM HUNTER\INPUT, OUTPUT"?
                      ↑                ()
*** Syntax Error: "(" expected instead of "\"
*** Syntax Error: Unexpected input -- ");" inserted to match "("
  40    VAR Q: INTEGER;
  41  BEGIN
  42  END.
```

Four Excellent Simple Recoveries.

```
  43  PROGRAM P(INPUT, OUTPUT);
  44    VAR I, PRIME, CHECK, NUMB: REAL, A:ARRAY\1 . . 6' OF REAL?
                                     ↑       ↑    ↑         ↑
*** Syntax Error: ";" expected instead of ","
*** Syntax Error: "[" expected instead of "\"
*** Syntax Error: "]" expected instead of " ' "
*** Syntax Error: ";" expected instead of "?"
  45  BEGIN
  46  END.
```

Excellent Scope Recovery.

```
  47  PROGRAM P(INPUT, OUTPUT);
  48  BEGIN
  49    REPEAT (* UNTIL LOOP IS FOUND *)
  50      X := 1
  51    UNTIL X = Y;
  52    X
        ↑
*** Syntax Error: "END." inserted to match "PROGRAM"
```

Excellent Simple (merge) Recovery.

```
  53  PROGRAM P(INPUT, OUTPUT);
  54  BEGIN
  55    BEGIN
  56      COUNT := 0;
  57      GO TO 2
            ↑
*** Syntax Error: "GOTO" expected instead of "GO" "TO"
  58      END;
  59  END.
```

Poor Simple Recovery.

```
  60  PROGRAM P(INPUT, OUTPUT);
  61    PROCEDURE FACTORIAL (A);
                              ↑
*** Syntax Error: "PROCEDURE" expected after this token
  62      VAR Q: INTEGER;
  63    BEGIN
  64      X := 1
  65    END;
  66  BEGIN
  67  END.
```

Two Excellent Simple Recoveries.

```
  68  PROGRAM P(INPUT, OUTPUT);
  69    VAR KEY, RECORD: ARRAY[1 .. LIMIT] IF AKFA;
                          ↑                ↑
*** Syntax Error: IDENTIFIER expected instead of "RECORD"
*** Syntax Error: "OF" expected instead of "IF"
  70  BEGIN
  71    X := 1
  72  END.
```

Excellent Simple (misspelling) Recovery.

```
  73  PROGRAM P(INPUT, OUTPUT);
  74  BEGIN REPEAT
  75    WRITELN('------------------');
  76    UNTILL EOF (INPUT);
            ↑
*** Syntax Error: Reserved word "UNTIL" misspelled
  77      X := 1
  78  END.
```

Ada

LISTINGS OF RUN WITH THE ADA SAMPLE ATESTS:
(General LR Version with level of deferral $k = 2$)

The three secondary recoveries, issuing diagnostics immediately following lines 23, 29, and 69, are rated as good. All other recoveries are rated as excellent.

```
  1   program atests is
          ↑
*** Syntax Error: "PROCEDURE" expected instead of "PROGRAM"
  2
  3
  4     x: float := 2.1 +;
                    ↑
*** Syntax Error: IDENTIFIER expected after this token
  5
  6     a: array INTEGER range 1 .. 10 of INTEGER;
              ↑                      ↑
*** Syntax Error: "(" expected after this token
*** Syntax Error: ")" expected after this token
  7     b: array [INTEGER range 0 .. 9] of FLOAT;
              ↑                      ↑
*** Syntax Error: "(" expected instead of "["
*** Syntax Error: ")" expected instead of "]"
  8     c: array (BOOLEAN);
                         ↑
*** Syntax Error: "OF IDENTIFIER" inserted to match "ARRAY"
  9
  10    type t is
  11      record
  12        a b: character;;
              ↑          ↑
*** Syntax Error: "," expected after this token
*** Syntax Error: Unexpected ";" ignored
  13      end record;
  14
```

```
 15     type b is INTEGER range 1 .. 30;
                          ↑
*** Syntax Error: Unexpected "INTEGER" ignored
 16
 17     subtype c is range 1 .. 30;
                     ↑
*** Syntax Error: IDENTIFIER expected after this token
 18
 19     proc count is
            ↑
*** Syntax Error: Reserved word "PROCEDURE" misspelled
 20         use TEXT_IO;
 21         x: integer;
                ↑
*** Syntax Error: "BEGIN" expected after this token
 22         GET(x);
 23         PUT(x);      -- a bad comment
                         ⟨--------------------⟩
*** Syntax Error: Unexpected input
 24     end count;
 25
 26     procedure q is seperate;
                           ↑
*** Syntax Error: Reserved word "SEPARATE" misspelled
 27
 28     procedure spell is
 29       b: array of float;
            ⟨--------⟩
*** Syntax Error: Unexpected input
 30         x: integer;
                ↑
*** Syntax Error: "BEGIN" expected after this token
 31         for i in 1 .. 10 loop
 32           b(i) := 0.0;
                  ↑
*** Syntax Error: "END LOOP;" inserted to match "LOOP"
 33     end;
 34
 35     function DAYS_IN_MONTH(M: MONTH IS_LEAP: BOOLEAN)
                                                  ↑                    return DAY is
*** Syntax Error: ";" expected after this token
 36     begin
 37       case M of
                ↑
*** Syntax Error: Unexpected input -- "IS WHEN" inserted to match "CASE"
 38           FEB ⇒ return 28;
 39           whan APR ⇒ return 30;
                ↑
*** Syntax Error: Reserved word "WHEN" misspelled
 40           when SEP | APR | JUN | | NOV ⇒ return 30;
                                      ↑
*** Syntax Error: IDENTIFIER expected after this token
 41           when others ⇒ return 31;
 42       end case;
              ↑
*** Syntax Error: "WHILE" expected after this token
 43
 44           z(y − 5 * j + k rem 7) > 0 loop
```

```
 45          x := x + 1;
 46          go to label;
                  ↑
*** Syntax Error: "GOTO" expected instead of "GO" "TO"
 47       end loop;
 48
 49       x := x + + 1;
                   ↑
*** Syntax Error: IDENTIFIER expected after this token
 50       y := ((3;
                   ↑
*** Syntax Error: ")" inserted to match "("
                   ↑
*** Syntax Error: ")" inserted to match "("
 51       return 28;
 52       << label
                   ↑
*** Syntax Error: ">>" expected after this token
 53       return 29;
 54
 55    end of DAYS_IN_MONTH;
             ↑
*** Syntax Error: Unexpected "OF" ignored
 56
 57    procedure p is
 58
 59       x: integer := 2
                        ↑
*** Syntax Error: ";" expected after this token
 60    begin
 61       loop
 62          if x > 0 then y := 2;
 63          if y < 0 then z := 3;
                                 ↑
*** Syntax Error: "END IF" inserted to match "IF"
                                 ↑
*** Syntax Error: "END IF;" inserted to match "IF"
 64       end loop;
 65    end p;
 66
 67    procedure test is
 68       x: array(123.144) of real;
 69       y: integer = 5;
                     ↑  ↑
*** Syntax Error: ":=" expected instead of "="
*** Syntax Error: statement part missing for unit
 70  begin
 71    if x(1) := y then
             ↑
*** Syntax Error: "=" expected instead of ":="
 72       null;
 73    elseif x(2) > y then
             ↑
*** Syntax Error: Reserved word "ELSIF" misspelled
 74       null;
 75    end if;
 76  end atests;
 77
```

ACKNOWLEDGMENTS

The authors wish to thank Frank DeRemer and the other referees for their careful reading and thoughtful criticisms and suggestions, which have improved the presentation and content of this paper.

REFERENCES

1. AHO, A. V., AND PETERSON, T. J. A minimum-distance error-correcting parser for context-free languages. *SIAM J. Comput. 1*, 4 (Dec. 1972), 305–312.
2. AMERICAN NATIONAL STANDARDS INSTITUTE. Ada programming language military standard. ANSI/MIL-STD-1815A, American National Standards Institute, Washington, D.C., Jan. 1983.
3. BURKE, M. G. A practical method for LR and LL syntactic error diagnosis and recovery. Ph.D. thesis, Dept. of Computer Science, New York Univ., 1983.
4. BURKE, M. G., AND FISHER, G. A. A practical method for syntactic error diagnosis and recovery. In *Proceedings of the SIGPLAN 82 Symposium on Compiler Construction* (June 23–25, 1982, Boston). ACM, New York, 1982, pp. 67–78.
5. BURKE, M. G. AND FISHER, G. A. A practical method for LR and LL syntactic error diagnosis and recovery. Res. Rep. RC 11111, IBM T. J. Watson Research, Yorktown Heights, N.Y., Mar. 1985.
6. CHARLES, P. Implementation of a LALR parser generator. Master's thesis, Dept. of Computer Science, New York Univ., 1982.
7. FEYOCK, S., AND LAZURUS, P. Syntax-directed correction of syntax errors. *Softw. Pract. Exper. 6*, 2 (Apr.–June 1976), 207–219.
8. GRAHAM, S. L., AND RHODES, S. P. Practical syntactic error recovery. *Commun. ACM 18*, 11 (Nov. 1975), 639–650.
9. GRAHAM, S. L., HALEY, C. B., AND JOY, W. N. Practical LR error recovery. In *Proceedings of the SIGPLAN 79 Symposium on Compiler Construction* (Aug. 6–10, 1979, Denver). ACM, New York, 1979, pp. 168–175.
10. JOHNSON, S. C. *YACC—Yet another compiler compiler.* Bell Laboratories, Murray Hill, N.J., 1977.
11. MAUNEY, J., AND FISCHER, C. N. A forward move algorithm for LR and LL parsers. In *Proceedings of the SIGPLAN 82 Symposium on Compiler Construction* (June 23–25, 1982, Boston). ACM, New York, 1982, pp. 79–87.
12. MICKUNAS, M. D., AND MODRY, J. A. Automatic error recovery for LR parsers. *Commun. ACM 21*, 6 (June 1978), 459–465.
13. PENNELLO, T. J., AND DEREMER, F. L. A forward move algorithm for LR error recovery. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Jan. 23–25, 1978, Tucson). ACM, New York, 1978, pp. 241–254.
14. PENNELLO, T. J., AND DEREMER, F. L. Practical error recovery for LR parsers. Unpublished Rep.
15. POONEN, G. Error recovery for LR(k) parsers. *Inf. Process. 77* (Aug. 1977), 529–533.
16. RIPLEY, D. J. *Pascal Syntax Errors Data Base.* RCA Laboratories, Princeton, N. J., Apr. 1979.
17. RIPLEY, G. D., AND DRUSEIKIS, F. C. A statistical analysis of syntax errors. *J. Comput. Lang. 3*, 4 (1978), 227–240.

Received May 1985; revised January 1986; accepted March 1986