ON THE ROLE OF ERROR PRODUCTIONS IN SYNTACTIC ERROR CORRECTION

by

C. N. Fischer

J. Mauney

COMPUTER SCIENCES TECHNICAL REPORT # 364

September 1979

On the Role of Error Productions In Syntactic Error Correction*

C. N. Fischer

J. Mauney

University of Wisconsin - Madison
Madison, Wi. 53706

*Research supported in part by National Science Foundation Grant MCS78-02570

Abstract

Error productions are presented as a means of augmenting syntactic error correctors. Such productions are able to simply and efficiently handle a wide variety of difficult error situations. Further, they can be employed without changing (or even knowing) the structure of the error correction algorithm being used. Examples of the use of error productions with the programming language Pascal are presented. Implementation and test results are included.

Keywords: Error correction, syntax errors, error productions, compiler design, diagnostic compilers, Pascal

1. Introduction

One of the most difficult aspects of the design of a syntactic error corrector is balancing the performance of a correction algorithm against its cost and complexity. If a fairly simple approach is chosen [3, 6, 8, 9, 17], error situations will arise in which the corrector makes a poor repair or even is unable to make any repair at all (skipping, e.g., out of the error context to restart parsing). If a more elaborate correction scheme is used [10, 11, 13-15], substantially better (but by no means perfect) performance can be obtained. However the cost of this improved performance can be high, requiring, e.g., changes to symbols already parsed [11] or examination of symbols far beyond the point at which an error is detected [11, 13-15]. Indeed, such schemes can impact the basic structure of a compiler, often precluding a simple, one-pass compilation scheme.

In this paper, we discuss the role of <u>error productions</u> in enhancing the performance of error correctors. An error production is simply a production added to a context-free grammar for the expressed purpose of recognizing (and sometimes correcting) a particular error. As an example, all Pascal programs must begin with a header statement (e.g., <u>program</u> p(input, output);). A way of anticipating the possible absence of this header is to include the following productions in a Pascal grammar*:

```
<Program> --> program <Id> ( <File List> ) ;
<Program> --> λ
```


The first production allows correct usage to be recognized while the second provides for an omitted header statement. Similarly, the Berkeley Pascal compiler [10] uses error productions to allow declarative sections (<u>const</u>, <u>type</u>, <u>var</u>, etc.) to appear in <u>any order</u>, accommodating a rather common (but erroneous) usage [16].

The idea of using error productions is by no means new; indeed, the correction technique of Aho and Peterson [2] is built entirely upon them. More recently, the YACC system [12] and the recovery technique of Granam, Joy and Haley [10] have made use of productions augmented with a special "error" token which allows parsing to be restarted in certain error situations.

Nevertheless, the use of error productions can pose special difficulties. If too many error productions are used, the resulting grammar can become unacceptably large or even <u>ambiguous</u>. If only a limited number of error productions are employed, the question of how (and when) to use them must be dealt with. The issues involved in choosing error productions for a semi-automatic error recovery technique for Pascal are discussed in [10]. In what follows, we elaborate upon this with special regard for the problems which arise in using fully automatic error correction algorithms.

---

*λ denotes the empty or null string.

Automatically generable error correction algorithms [3, 6, 8, 9, 17] have many attractive properties. They are simple to use (often basing corrections on an abstract "costing" model), fast (typically requiring only a few milli-seconds per correction) and quite powerful (with typically 85-90% of their corrections rated as either "good" or "excellent"). Nevertheless, a number of fundamental assumptions are commonly built into these correctors:

(1) Since most modern parsing techniques (SLR(1), LALR(1), LL(1),etc.) guarantee that symbols accepted by a parser must form a correct prefix of some program, correctors commonly assume that once a symbol is accepted it will never be changed*

(2) The validity (or at least plausibility) of a proposed correction is determined by examination of only a fixed number (commonly 1 but sometimes more [10,17]) of the remaining input symbols.

(3) Context-sensitive rules (e.g., type and scope rules) are ignored in determining a correction.

---

*Note that this allows translation of a symbol to occur once it is accepted without fear that semantic actions may later have to be "undone".

These assumptions are very useful in organizing and simplifying the correction process. Nonetheless, the limitations to the correction process which they impose inevitably lead to a number of "poor" repairs which are difficult to root out.

It is often the case that these sorts of problems can readily be handled by the judicious use of error productions. Indeed, such usage is especially attractive in that normally no changes are needed in the corrector itself or the host compiler. Rather, the only overhead involved is the extra parse table space required and the possible addition of a few new semantic routines for the error productions. This makes adding (and if necessary removing) error productions particularly simple and thus makes experimentation with them an almost trivial matter. This is in sharp contrast to changing an existing error corrector which often is a forbiddingly difficult undertaking.

In the following sections, we will illustrate the use of error productions in handling some very common (but difficult) error situations. We will use Pascal in our examples, but the techniques are applicable to virtually any programming language. Some of the examples presented are in actual use with UW-Pascal, a diagnostic Pascal compiler [7]. The others have been tested with the corrector described in [6] to gauge their cost and effectiveness. In all cases, the examples described appear to be

very useful adjuncts to the kinds of error correctors used in practice.

2. The Pascal "; else" Problem

A very common error in Pascal (and most other Algol-based languages) is the incorrect use of a ';' as a statement terminator rather than a statement separator [16]. Thus the following illustrates what we shall term the "; else" problem:

```
if A = 1 then write(1);
else write(2);
```

The ';' preceding the else is illegal and the obvious (and apparently trivial) correction is to delete it.

Rather surprisingly, this is quite a difficult correction to realize. Consider the following grammar fragment drawn from an LALR(1) grammar for UW-Pascal. It is representative of how Pascal if statements are generated.


```
G1: <Stmt List>   --> <Stmt List> ; <Stmt>
                   |  <Stmt>

    <Stmt>        --> <Open Stmt>
                   |  <Closed Stmt>

    <Open Stmt>   --> if <Expr> then <Stmt>
                   |  if <Expr> then <Closed Stmt> else <Open Stmt>

    <Closed Stmt> --> if <Expr> then <Closed Stmt> else <Closed Stmt>
                   |  Other
```

Note that <Open Stmt> and <Closed Stmt> are used to handle dangling else's unambiguously. If we try to parse ...if <Expr> then Other; else..., we have a problem in that this is reduced to ...<Stmt>; else... before the else is scanned and the error is detected. This is a real problem in that we would have to "undo" reductions and semantic actions done before the error was detected to effect the desired correction. This difficulty could be dealt with by using a two symbol lookahead parser, but virtually all parser generators in common use disallow multiple symbol lookaheads. In any event, the increased parser complexity and table sizes which would result seem quite unwarranted. Alternately, the scanner could be modified to scan an extra token once a ';' is found (to see if an else follows). This however is again unattractive in that it requires a special modification to the scanner to handle a single error. Further, the scanner (which should be simple and fast) is made more complex (it must buffer an extra token), slower (it must scan ahead after all ';'s), and more difficult to use (since it must somehow be able to signal a syntax error and invoke error correction).

Thus the use of error productions seems indicated. An obvious approach is to add the productions:

<Else> --> ; else | else

<Else> would then be substituted for else in G1. Now however we have a shift-reduce conflict in the context of ...if <Expr> then <Closed Stmt> ; else... with a lookahead of ';'. (<Closed Stmt> can be reduced to <Stmt> or ';' can be read as part of "; else"). This illustrates the care needed in choosing error productions. An alternate (and far less obvious) approach is to modify G1 so that ';' is included on the right hand side of various productions generating statements:

```
G2: <Stmt List>     --> <Stmt List ;> <Stmt>  |  <Stmt>

    <Stmt List ;>   --> <Stmt List ;> <Stmt ;>  |  <Stmt ;>

    <Stmt ;>        --> <Open Stmt ;>  |  <Closed Stmt> ;

    <Open Stmt ;>   --> if <Expr> then <Stmt ;>
                     |  if <Expr> then <Closed Stmt> <Else> <Open Stmt ;>

    <Stmt>          --> <Open Stmt>  |  <Closed Stmt>

    <Open Stmt>     --> if <Expr> then <Stmt>
                     |  if <Expr> then <Closed Stmt> <Else> <Open Stmt>

    <Closed Stmt>   -->
                    if <Expr> then <Closed Stmt>  <Else> <Closed Stmt>
                     | Other

    <Else>          --> else  |  ; else
```

Now in ...if <Expr> then <Closed Stmt> ; else..., we read the ';' then use lookahead to decide whether ';' is part of "<Closed Stmt> ;" or part of "; else". The error productions are vital in forcing the use of lookahead at this point.

Although 8 new productions are added, the corresponding increase in parse table size is quite minor (in the UW-Pascal grammar, only 48 table entries,* requiring 96 bytes were added). Also, the recognition of the error production "<Else> --> ; else" provides an excellent opportunity (exploited by UW-Pascal) to issue a specific diagnostic explaining why this usage is incorrect. The use of error productions in this case was well justified because:

(1) We wished to guarantee correct treatment of a rather common error.

(2) Normal correction techniques failed because the error was detected too late.

3. Relational Expression Errors

Another case amenable to the use of error productions arises from Pascal's unconventional operator precedence levels. In particular, relational operators are applied last (after all other levels). Thus "A = B or C" is parsed as "A = (B or C)". Further, relational operators don't associate so "A = B = C" is

---

*In these measurements, only non-error entries in the parse table are counted since a compressed table format is commonly employed.

syntactically illegal, as is "A = B or C = D". Experience has shown that Pascal users often forget to parenthesize relational expressions (to force operator precedence) and thus we often see the following sort of incorrect usage:

if A = B or C = D then ...

When we attempt to parse this, we obtain:

if <Expr> = D then ...

In such circumstances, correctors which require only one correct token after error repair often correct this by deleting the '=' and then inserting then to obtain:*

if <Expr> then D then...

In this case a second, cascaded error will occur after D is read. Indeed, even those techniques which require that more than one legal token be found after error repair will fall prey to an analogous error:

if <Expr> = D [...] then ...

Here the subscript list qualifying D can be arbitrarily long and can thus "shield" the then which follows. In fact even those error correctors which, via a "forward move", can always correct such errors are not entirely satisfactory because they give no insight into why the error occurred. That is, expressions such as ...A = B or C = D... look correct and in many languages are correct. The error involved is really semantic, not syntactic, and a syntax error alone does not convey the real nature of the

*Such a repair is almost forced because then is the only token which can follow the <Expr> in this context.

problem. Thus such errors are particularly suited to the use of error productions.

To handle such errors, UW-Pascal uses error productions which allow relational operators to associate. These error productions consume the remainder of an illegal expression and cue the generation of a special diagnostic message. Thus the production

<Expr> --> <Simple Expr> <Rel Op> <Simple Expr>

is expanded into:

```
G3: <Expr>     -->  <Rel Expr>
                 |  <Rel Expr> <Flag> <Rel Tail>

    <Rel Expr> -->  <Simple Expr> <Rel Op> <Simple Expr>

    <Flag>     -->  λ

    <Rel Tail> -->  <Rel Op> <Simple Expr>
                 |  <Rel Tail> <Rel Op> <Simple Expr>
```

The production "<Flag> --> λ" has a semantic routine which issues a syntax error message (since relational operators don't really associate) as well as a reminder that, unless parenthesized, relational operators are applied last. This reminder is especially helpful since the user almost certainly got an unexpected (and otherwise confusing) type error (since, e.g., "A = B or C = D" is parsed as "A = (B or C) = D" and B and C probably are not Boolean).

Again, although 5 productions have been added, the extra costs involved are small (for the UW-Pascal grammar, 50 new parse

table entries, requiring 100 bytes). Note too that we do not correct "A = B or C = D" into "(A = B) or (C = D)". This, unfortunately, is all but impossible to do because "A = B or C" is syntactically legal and is parsed as "A = (B or C)". By the time the second '=' is found, it is too late to undo the semantic processing of "A = B or C".

4. Type and Scoping Errors

Even the best of error correctors can make poor repairs when corrections are based purely on syntactic grounds. Thus a corrector may choose an apparently excellent correction and then have it immediately rejected once compilation (and semantic processing) are restarted. Consider the following error situation: ...A B... This may plausibly be corrected into:

```
...A := B ... or,
...A ;  B...  or,
...A (  B...  or,
...A [  B...
```

The choice of which alternative is best is really dependent on the types of A and B and this information is usually ignored in the correction process. In some cases, a forward move (which examines additional right context) can aid in the choice of corrections. Thus given ...; A B] := 1;..., an insertion of a '[' after the A is strongly suggested. However, if A is not an

array, the suggested correction should be rejected (to avoid a spurious semantic error).

A few error correction techniques have been studied which use semantic information. For example, Feyock and Lazarus [5] and Graham, Haley and Joy [10] allow semantic routines to be called from the error corrector to determine if a proposed correction is semantically valid. Dion [4] suggests the use of an attributed grammar to represent context—sensitive information and extends context-free parsing and correction algorithms to utilize this information.

These approaches are fairly effective but some care is required. For example, any semantic routines called by an error corrector must be side-effect free (since the proposed corrections are only tentative and may be rejected). Further, these approaches require that the error corrector itself know about context-sensitive issues (e.g., which semantic routines or attributes to use). It may not be easy (or desirable) to add such capabilities directly to an extant corrector.

A possible alternative is to use error productions to encode certain kinds of context-sensitive information. An especially simple way of doing this is to replace the terminal symbol <Id> (which represents identifiers) by a number of distinct terminals representing various kinds of possible identifier classes. In

our experiments we used 5 distinct terminals: <Array Id>, <Proc Id>, <Fct Id>, <Other Decl Id>, <Udecl Id>. <Array Id> represents identifiers of type array; <Proc Id> and <Fct Id> represent procedure and function identifiers. <Other Decl Id> represents any other kind of declared identifier (record, pointer, type, scalar, etc.) while <Udecl Id> represents an undeclared identifier. Note that the kind of an identifier can easily be determined by a scanner via a simple symbol table lookup*.

Given these new kinds of identifiers, all occurrences of <Id> in a grammar are replaced by one or more of the new identifiers. Thus an identifier on the left-hand side of an assignment is replaced (in Pascal) by <Array Id>, <Fct Id> and <Other Decl Id> (i.e., all three choices are allowed in an assignment statement). Similarly, in an array qualification, only <Array Id> is allowed, and in a function or procedure call, only <Fct Id> and <Proc Id> may appear. In a declaration, any kind of identifier is allowed (since any identifier may be redeclared†).

---

*In fact, many scanners do a symbol table lookup anyway to avoid returning a character string to the parser and semantic routines.

†Semantic routines still verify that the redeclaration is in fact legal.

Note that <Udecl Id> is useful in preventing an undeclared identifier from being accepted in a context where only a declared identifier is legal. Similarly, in correction schemes which rely on a cost criterion to choose repairs, it is easy to make undeclared identifiers cheaper to delete than declared identifiers. This helps to improve the overall quality of corrections since a declared identifier is more likely to be correct in an error context than an undeclared identifier.

Choosing a correction for ...A B... is now a good deal easier since A and B are represented by terminals encoding some scope and type information. Thus if A is an integer (and therefore is scanned as <Other Decl Id>), neither '[' nor '(' can follow it, while if A is an <Array Id>, a '[' is a plausible insertion. So too, if B is a <Proc Id>, insertion of a ';' can occur while if it is a <Udecl Id>, it may well be deleted. We can even include type information in nonterminals. For example, in the UW-Pascal grammar, a '[' can follow an <Array Var> but not an ordinary <Var>. An <Array Var> can obviously generate an <Array Id>. Now when a right-hand side "<Var> ↑" is processed, it is reduced to either <Array Var> or <Var> depending on lookahead. However it is possible to have only a single production "<Var> --> <Var> ↑" (which doesn't need lookahead). The semantic routine for this production can then replace <Var> by <Array Var> (e.g., by replacing the top LALR(1) parse stack state) if the corresponding data object is an array. This then

allows type information to be used in the correction of, e.g., ...R.A.B↑ C...

This approach is attractive in that it is easy to use (only the scanner and the underlying grammar need be changed) and readily extendable (we can add new error productions whenever we desire). Further, it isn't overly expensive: In adding the 5 identifier kinds described above to UW-Pascal's grammar, we added only 11 productions (to a grammar of 257 productions). This resulted in 322 new parse table entries requiring 644 bytes.

5. Missing Statement Headers

Many (indeed most) error correctors have a great deal of difficulty with missing statement headers. This is exemplified by the infamous "missing <u>if</u>" problem:

...;A = B <u>then</u> Write(1) <u>else</u> Write(2);...

The problem here is twofold. First, since parsers consume input as long as possible, it is common for the leftmost identifier of an expression exposed by a missing header to be consumed as the left-hand side of an assignment statement. Thus repairs of the following sort are often produced:

...;A := B <u>then</u> ...

This of course leads to later, cascaded errors and is therefore quite undesirable. It is possible to back up over an accepted token <u>if</u> no semantic actions have occurred [10], but in some cases this can't be done (e.g., in ...; A[1] = B <u>then</u>...).

A second problem arises from the fact that <u>many</u> <u>different</u> statements begin with a distinctive header followed by an expression. Thus ...; A = B... might be a malformed <u>if</u> statement, <u>while</u> loop, <u>case</u> statement or assignment statement. The only way to decide is to process the entire expression to see what follows it. That is, a forward move (in effect) is needed.

A remarkably simple way of dealing with missing statement headers is to introduce error productions which <u>anticipate</u> such omissions. Thus we can generate an <If Head> (which begins an <u>if</u> statement) as:

<If Head> --> <u>if</u> <Expr> <u>then</u>
| <Expr> <u>then</u>

Similar productions can be used to produce a <While Head> and a <Case Head>. The introduction of such productions forces lookaheads to be examined before any reductions occur, so there is no danger that part of an expression will be incorrectly parsed as the left-hand side of an assignment. Further, these error productions are not applied until an entire expression (no matter how large) is reduced, so the effect of a forward move is obtained without any special effort. Naturally, the semantic

routines associated with such error productions can be used to generate special diagnostics detailing the problem.

Note that malformed assignment statements of the form ...<Id> <Expr>... or ...; := <Expr>... can readily be handled by a corrector. However an error of the form ...; <Expr> ;... can present problems since it can be corrected into either ...; <Id> := <Expr> ;... or ...while <Expr> do ;... Using the above error productions, the latter correction is likely to be produced since <Expr> will be accepted as a prefix of one of these error productions before an error is detected. We can handle this case by adding an assignment statement error production "<Stmt> --> <Expr>". This allows us to insert "<Id> :=" if we wish or alternately to delete the <Expr> (e.g., by jumping around it or by ignoring its result). An interesting ambiguity does however arise. Is, e.g., ...; h(i);... a procedure call or a malformed assignment inolving a function? The question hinges on whether 'h' is a procedure or a function. This can clearly be checked by the appropriate semantic routines (once h(i) is fully reduced), but for correction purposes, the techniques of section 4 are very useful: we simply have the scanner distinguish between procedure identifiers and function identifiers. Then h(i) is easily (and automatically) resolved and the appropriate corrections (if any) can be initiated.

The syntax of the header of a Pascal for loop is typically generated by:

<For Head> --> for <Id> := <Expr> <To or Downto> <Expr> do

This header almost (but not quite) imbeds a Pascal assignment statement. The difference is that only a simple identifier (and not, e.g., a field of a record) can be used as a loop index. This restriction is often misunderstood and the syntax error obtained when a qualified identifier is used as a loop index can be quite obscure. This suggests generalizing the for header's syntax to:

<For Head> --> for <Lhs> := <Expr> <To or Downto> <Expr> do

The restriction on the form of <Lhs> can be checked semantically and, if necessary, a very specific disgnostic can be issued. Also, it then becomes possible to allow (as an extension, with a suitable warning) a qualified identifier to be used as a loop index.

Once this change is made, a missing for header is handled by adding:

<For Head> --> <Lhs> := <Expr> <Msg> <To or Downto> <Expr> do
<Msg> --> λ

The semantic routine for "<Msg> --> λ" is used to issue the necessary diagnostic.

Finally, we consider briefly the problem of a missing repeat header. This error is very difficult to handle well as we don't

even discover the error until an unexpected until is found. Ordinary correction techniques might insert a repeat just ahead of the until (and in doing so might create an infinite loop) or they might delete the until and thus reduce the error to the ...; <Expr>;... case discussed above. The latter correction is preferable, especially if ...; <Expr>;... is then deleted. Note that the use of an error production "<Stmt> --> until <Expr>" will cause a parsing ambiguity (with "<Stmt> --> repeat <Stmt List> until <Expr>") when a correct repeat loop is parsed. This can, however, be handled by a parser generator such as YACC which allows ambiguous constructs to be parsed [1]. If both productions are applicable, we specify that the repeat loop production is to be chosen.

The above error productions have been tested with the error correction technique described in [6]. They appear to handle missing statement headers quite well. Further, they are quite inexpensive to use: Only 5 error productions were needed to accommodate missing if, case, while and for headers*. These productions added 319 new parse table entries, requiring 638 bytes.

As a final experiment, all of the error productions described above were added to the UW-Pascal grammar (including

*The error production "<Stmt> --> <Expr> was excluded because it necessitated the use of the terminals <Proc Id> and <Fct Id>.

"<Stmt> --> <Expr>"). This represented a total of 34 new productions and increased the size of the grammar to 291 productions. The corresponding increase in parse table size was 958 entries* (requiring 1916 bytes) to a final total of 3200 entries (and 6400 bytes). When the necessary semantic routines for error productions and changes to the scanner (to produce new identifier classes) are included, an overall increase of 2500-3000 bytes can be expected. However given the wide range of rather difficult error situations which these productions address, this increase is quite modest. Further, because almost all changes are localized to the grammar itself, implementing the enhancements represented by such productions is almost trivial -- new parse tables are created (using a parser generator) and a few new semantic routines are added. This is in very sharp contrast to direct changes to an error corrector which can be extremely slow and difficult to implement (since, at a minimum, operation of the corrector must be thoroughly understood).

*This increase is greater than the sum of the individual increases cited above. This is due to the addition of "<Stmt> --> <Expr>" as well as the mutual interactions of the new productions.

Summary

A number of examples illustrating the use of error productions in augmenting the performance of syntactic error correctors have been presented. Such productions are able to cope with a wide variety of error situations beyond the scope of most common error correctors. Such error situations include missing statement headers, type and scoping errors, and errors normally detected too late to be effectively repaired.

The costs involved in using error productions are normally quite modest and usually involve only a small increase in parse table size and the addition of a few new semantic routines (to issue necessary diagnostics). Very spectacular savings can be realized in the time needed to implement error corrector changes via error productions. Because the error corrector being used does not need to be redesigned or modified, changes are localized to the grammar itself. This makes extension of the error corrector particularly easy and safe (at worst, the extended grammar can be rendered unparsable).

The examples of error production usage cited are in no sense complete, nor are they intended to be. One of the prime virtues of this approach is its ability to handle new and unexpected error situations simply and efficiently. Because error productions can be added to a grammar <u>without changing</u> (or <u>even</u>

<u>knowing</u>) the structure of the associated error corrector, such productions are a valuable adjunct to the correction process. They are, in effect, a "second line of defense" which can be utilized, when necessary, to improve the performance of a corrector. Viewed in this light, error productions will certainly have a continuing role in the construction of high-quality syntactic error correctors.

References

1. A.V. Aho, S.C. Johnson and J.D. Ullman, Deterministic parsing of ambiguous grammars, Comm. ACM 18, 441-452 (1975).

2. A.V. Aho and T.G. Peterson, A minimum distance error correcting parser for context-free languages, SIAM J. Comput. 1,4, 305-312 (1972).

3. S.O. Anderson and R.C. Backhouse, Least-cost error recovery in LR parsers: a basis, Heriot-Watt University, Edinburgh (1979).

4. B.A. Dion, Locally least-cost error correctors for context-free and context-sensitive parsers. University of Wisconsin-Madison, Ph.D. thesis, Tech. Report 344, (1978).

5. S. Feyock and P. Lazarus, Syntax-directed correction of syntax errors, Software - Practice and Experience 6, 207-219 (1976).

6. C.N. Fischer, B.A. Dion and J. Mauney, A locally least-cost LR-error corrector, Univ of Wisconsin Technical Report 363. Submitted to ACM Trans. on Programming Languages and Systems, (1979).

7. C.N. Fischer and R.J. LeBlanc, UW-Pascal reference manual, Madison Academic Compting Center, Madison, Wi., (1977).

8. C.N. Fischer, J. Mauney and D.R. Milton, A locally least-cost LL(1) error corrector. To be submitted to IEEE Trans. on Software Engineering.

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

10. S.L. Graham, C.B. Haley and W.N. Joy, Practical LR error recovery, Proc. of the Sigplan Sym. on Compiler Construction, in Sigplan Notices, 14, 8, 168-175 (1979).

11. S.L. Graham and S.P. Rhodes, Practical syntactic error recovery, Comm. ACM 18, 639-650 (1975).

12. S.C. Johnson, YACC - yet another compiler compiler, Bell Laboratories, Murray Hill, N.J., (1977).

13. M.D. Mickunas and J.A. Modry, Automatic error recovery for LR parsers, Comm. ACM 21, 459-465 (1978).

14. T.J. Pennello and F.L. DeRemer, A forward move algorithm for LR error recovery, Proc. Fifth Ann. ACM Symp. on Principles of Programming Languages, 241-254, (1978).

15. D.A. Poplawski, Error recovery for extended LL-Regular parsers, Purdue University, Ph.D. thesis, (1978).

16. G.D. Ripley and F.C. Druseikis, A statistical analysis of syntax errors, Computer Languages 3, 227-240 (1978).

17. K.C. Tai, Syntactic error correction in programming languages, IEEE Trans. on Software Engineering Vol. SE-4, 5, 414-425 (1978).