Practical LR Error Recovery†

Susan L. Graham
Computer Science Division
University of California, Berkeley
Berkeley, Ca. 94720

Charles B. Haley
Bell Laboratories
Murray Hill, NJ 07971

William N. Joy
Computer Science Division
University of California, Berkeley
Berkeley, Ca. 94720

Abstract

We present a practical, language independent mechanism for error recovery in LR parsers. The method is easy to implement in existing parser generators. It uses only the normal parse tables and a small amount of symbol cost information. It is possible to use compressed parse tables with the method, as well as other LR augmentations such as precedence and associativity declarations. The method is efficient enough to be used in environments where space and time are at a premium.

Our method utilizes a number of previous error recovery ideas: forward moves in the input after detection of errors to incorporate right context, recovery choice based on weighted costs, and the use of special productions to indicate major productions in the grammar for error recovery. The method also makes use of semantic information in choosing between possible recovery actions.

We have implemented the method in a production Pascal translator which has been in use for instruction at a number of universities for two years. We report here on measurements of our system on a standard data base of Pascal programs with errors.

1. Introduction

There have been a large number of papers in recent years on theoretical aspects of error recovery, among them [GR 75], [PDeR 78], and [MM 78] (which gives other references), but noticeably few actual implementations have been reported in production use. The gap between theory and practice may be attributable to many factors, not the least of which is that the error recovery mechanisms tend to be large and slow. Compiler writers and compiler users have traditionally been more concerned about the quality of the generated code and the speed of the compilation than the quality of error diagnostics.

This paper discusses a technique for error recovery which is practical in both space and time and which can be implemented in existing parser generator systems and on small machines. The method presented in this paper represents a series of compromises between power and practicality. Unlike some other recent studies attempting to extend error recovery ideas to LR parsing ([DR 76], [MM 78], [PDeR 78] for example) we have selectively implemented those techniques which could be treated efficiently. In gaining speed and saving space, we have lost the capability to provide insight into certain kinds of errors. It is thus essential that our method recover gracefully in the presence of errors which are, after analysis, recognizably beyond the power of our method to correct.

The Pascal system in which our error recovery is incorporated [JGH 79] has been in heavy daily use for over two years by people ranging from beginning programming students to experienced and sophisticated programmers. User reaction to the system has been extremely favorable. Their reaction is substantiated by the performance of the method on sample programs gathered from students in programming classes and from other researchers.

2. A sample run

We begin with a simple Pascal program which has appeared in the literature before [GR 75], and which illustrates both the strengths and the limitations of our method. Figure 1 is the output resulting from an attempted translation of this incorrect program. The line numbers in the figure are provided by the translator and are not part of the program text.

Our translator flags syntactic errors with a line of dashes and an arrow marking the point in the line where the correction is made. Errors flagged with an "E" are hard errors which could not be repaired, while errors flagged with an "e" have been repaired. If no hard errors occur during translation, code will be generated.

Our method uses context past the point of error in choosing a recovery action. When an error is encountered we scan ahead, suppressing the listing. We associate with each token enough information to locate it accurately in the source text for a possible error diagnostic. After a recovery action is chosen, the implied modification of the upcoming input is performed, and

† Research supported by NSF Grant MCS74−07644−A03 and by an IBM Graduate Fellowship to W. N. Joy

Figure 1 — a sample run

```
Berkeley Pascal -- Version 1.0 (January 18, 1978)

Sat Jan 20 19:55 1979  err.p

     1  program cacm(input,output);
     2  label 1,2,3;
     3  var a,b: array[1..5 1..10] of integer;
e ------------------------┤--- Inserted ','
     4       i,j,k,l: integer;
     5  begin
     6       3: i + j > k + l * 4 then go 1 else k is 2;
e -------------┤--- Inserted keyword if
E ---------------------------------------┤--- Malformed statement
E ------------------------------------------------┤--- Malformed statement
     7       a 1,2 := b [ 3 * ( i+4,j*/k]
e -------------┤--- Inserted '['
e ----------------┤--- Inserted ']'
e --------------------------------┤--- Inserted ')'
E ----------------------------------┤--- Inserted identifier
     8       if i=1 then then goto 3;
e ----------┤--- Inserted ';'
e ----------------------┤--- Deleted keyword then
     9  2: end.
In program cacm:
  E - label 1 was declared but not defined
```

parsing resumes over the unused lookahead remaining from the error recovery and then the rest of the input stream.

Recovery actions are of two kinds. The most common changes, termed "first level recovery", are single token insertions, deletions and replacements, such as the insertion of the token "," on line 3. If no simple, single terminal, change is possible, then a second level recovery is attempted.

Second level recovery actions involve the detection of a badly formed phrase, which approximates, but does not match, a sequence of tokens derivable from a nonterminal of the grammar. For example, on line 6, the replacement of the identifier *go* by the keyword goto is rejected because of the proximity of another error. The recovery mechanism recognizes that it has a large part of a malformed statement and skips over the token "1", effectively replacing go 1 with a statement.

We discuss this example further in later sections of the paper

3. Basic principles of the method

The key to the efficiency and quality of the method is the restriction of the changes considered to a simple set which contains a large percentage of common programming errors. By using context, symbol costs, and semantic information to limit the possibilities, we can afford to perform a forward move over a bounded number of terminals for each proposed change.† Since the context information is implicit in the parse tables, its use occurs naturally.

An important property of the classes of changes the method chooses is that they are easily conveyed to the user. Since syntax errors are often reflections of misunderstandings of the language, it is important to point out clearly to the user how the translator sees the error. It is very important that all diagnostics are in terms of the source program with the point of error clearly marked. Single symbol changes can be presented to the user without reference to the inner workings of the grammar; second level recovery actions deal with components of the language such as expressions, statements and declarations with which even beginning programmers are familiar.

Our goal is not to repair the program, but to continue parsing and produce a maximum number of useful diagnostics with a minimum of noise. The diagnostics we wish to produce are judged by their usefulness to the user. If the system cannot repair an error with a simple change, we prefer to say so in one message, rather than producing a flurry of small changes which are unlikely to be correct. In order to achieve high quality at low cost, we do not restrict ourselves to "pure syntax" recovery but use lexical and semantic considerations as well. It has proven useful in some cases described subsequently to implement certain syntactic restrictions of the source language by semantic checks.

† Throughout this paper we use the term "semantic" in the usual compiler sense.

4. The method

4.1. The parser

Our method is implemented in the environment of the compiler generator of Aho and Johnson called YACC [Johnson 77], but can be easily incorporated in other LR parser generators. This parser generator uses a LALR(1) technique, although the method should work equally well with LR and SLR generators.†

Our method allows most space saving techniques for parse table compression to be used, but makes some simple but critical demands on the way in which information is compressed in the parse tables. It requires that sufficient information exist there to prevent "too many" reductions from occurring before an error in which the next input token will not be shiftable is detected.

Recall that a syntax error is detected in an LR parser if, in the current state, there is no shift or reduce move indicated for the current input token. In order to save space many SLR(1) and LALR(1) table generators enumerate lookahead symbols only to the extent necessary to resolve parsing action conflicts. In particular, if a state has exactly one reduction (completed rule) that reduction is indicated for all input tokens which are not shiftable. These "default reductions" reduce the tables to acceptable size, but delay error detection because reductions are made without a legality check on the input token. In our initial work to improve the error recovery in an early version of the Pascal system, we discovered that these additional reductions severely hampered the recovery.

Consider the following fragmentary input:

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

We have indicated with a '?' here that an error exists between the identifiers b and c. The crucial point to observe is that "a := b" may well correspond to a complete statement in our language grammar. If our parser generator compresses lookahead to the extent that it allows "a := b" to reduce to a statement, then we will have no chance to repair this statement by the simple and obvious change, inserting a '['.‡

Eliminating all default reductions would give unacceptably large tables. Instead, we replace default reductions by the actual lookahead symbols only in specially designated states. If those contexts are chosen appropriately, the selective expansion of lookaheads prevents too many reductions from occurring, while achieving good space compression.

We have borrowed from YACC a notion of a special **error** token which is recognized by both the table generator and the parser. As in YACC, our grammars can contain productions, called "error productions" which contain the **error** token as one of the tokens in the rule. The table generator constructs the table in the usual way, except that it enumerates all lookaheads in states which have a shift action on the **error** token.

When the parser detects an error, the **error** token is inserted before the actual input token. The parser reduces the stack until it reaches a state in which it must inspect the lookahead symbol. It then "sees" the **error** token and invokes the error recovery mechanism.

4.2. The grammar and the error productions

Our grammar was adapted from the one given in the Pascal User Manual and Report [JW 75] by removing ambiguities, using a simplified expression grammar augmented by precedence and associativity declarations, and making the changes described in section 4.4. In order for selective lookahead expansion to work well, and to provide hooks for the second level recovery mechanism (described below), the major nonterminals of the grammar generate error productions. Those portions of the grammar involving the error productions are given in Figure 2.

As the reader can see from Figure 2, the error productions cause lookahead expansion following recognition of the program token, statements, expressions, identifiers which begin statements or expressions, and (perhaps partial) lists of constant definitions, type definitions, variable declarations, declaration sections, record fields, and record variant fields. These are the major units in Pascal, and it is important that the error recovery mechanism take hold to deal with errors following them.

The error productions, which are critical in keeping the parse tables small while preventing too many reductions from taking place, also guide the second level error recovery. This use will be discussed subsequently.

4.3. First level recovery algorithm

We next describe the process of choosing a first level recovery action at the point of error. We have already seen how, in the presence of an error, default reductions can occur. The default reductions are analogous to the backward move of Graham and Rhodes and serve to condense the left-context information.† The error productions in the grammar control how much left context condensation occurs.‡

After condensing the left context information, we form a set of feasible first level corrections without a preliminary forward move. This strategy is unlike the Graham and Rhodes approach. Given the apparently large cost in space and time of LR forward moves [DR 76], [Modry 76] we chose to attempt to find a small but rich set of changes and to run a forward move on each. While this strategy is potentially slower than a single preliminary forward move, it has the advantage that we can use

† We will assume that the reader is familiar with LR and LALR parsing. Readers unfamiliar with LALR parser table construction are referred to [AU 77] for a excellent and thorough presentation.

‡ We do not consider undoing reductions to be a feasible approach in the context of our error recovery.

† Our method is based on the work of Graham and Rhodes [GR 75]. We use their terminology in the remainder of this paper.

‡ Note that control over the amount of left context condensation is not possible in the Graham-Rhodes method.

Figure 2 – error productions

```
programHeader ::= program id ( idList ) ; | program error

declarations ::= declarations decl | decl | declarations error

decl ::= labelDecl | constDecl | typeDecl | varDecl

constDecl ::= const id = constant ;
            | constDecl id = constant ;
            | const error
            | constDecl error

typeDecl ::= type id = type ;
           | typeDecl id = type ;
           | type error
           | typeDecl error

varDecl ::= var idList = var ;
          | varDecl idList = var ;
          | var error
          | varDecl error

recordFieldList ::= fixedPart variantPart

fixedPart ::= field | fixedPart ; field | fixedPart error

field ::= λ | idList : type

variantPart ::= λ
              | case id of variantList
              | case id : typeId of variantList

variantList ::= variant | variantList ; variant | variantList error

variant ::= λ
          | constList : ( fieldList )
          | constList : ( )

caseStList ::= caseListElement
             | caseStList ; caseListElement
             | error
             | caseStList error

statement ::=   ... | id error | error

expression ::=  ... | id error | error
```

semantic information to aid in the recovery selection process without worrying about reductions occurring out of order. In many cases, such as in line 7 of Figure 1, semantic information is as good as right context in pruning the set of possible changes. We will return to this example and the issue of semantic information in the next section.

The first level changes considered by the error recovery routine are single token changes using the *shift* tokens in the current state and the input token that caused the error to be detected. The possible changes are deletion of the input token, insertion of each legal shift entry before the input token, and replacement of the input token by each legal shift entry.

Consider the first error in the example in Figure 1. The parser is in state 275 when the error is detected. This state has the form:

structType ::= `array [` *simpleTypeList* • `]` `of` *type*
simpleTypeList ::= *simpleTypeList* `,` • *simpleType*

,	shift to state 305
]	shift to state 304
default	declare error

In our example, the input "1..5" was reduced to *"simpleType"* although a full LR parser would have declared error because the second input "1" was not a legal symbol following the "5".

In order to try to use the symbols "," and "]" to perform a repair, we first scan ahead a bounded number of tokens (five tokens in our production translator used for this example) to prepare for the forward moves. Each possible change (except, as explained subsequently, those that can be immediately eliminated) is tried by running the translator forward over this input stream until we either run out of lookahead tokens, encounter another syntax error, or perform a reduction which semantically indicates the presence of another error.† During this forward move we are running only a subset of the semantics of the compiler, avoiding any permanent changes to its data structures since the parsing is over tentative forms of the input.

In addition to changes based on the input token that signalled the error, if the last action before the error detection is a shift, in which case the previous input symbol has not yet been reduced, the recovery routine backs up the input one token and considers the same kind of repairs. By limiting the backup to input tokens, we avoid the need to undo semantic actions.

In choosing the best first level repair we have adopted the basic principles of the Graham and Rhodes method and assigned costs to each terminal symbol. The basic cost of a repair is the sum of the costs of the tokens involved.

To incorporate information from the forward move, we multiply each basic cost by a factor to indicate lack of progress into the forward context. If the repair allows us to consume all of the forward context then this factor is unity. Consuming less of the input gives higher multiplicative factors. Thus if the basic cost of a repair prior to the forward move is greater than the total cost of one of the changes which we have already found, then the repair is not even potentially "better". This observation allows us to eliminate many potential repairs without a forward move.

If the symbol being replaced, deleted or inserted, such as a number or an identifier, has a semantic attribute, then the repair is assigned higher cost. As in Graham and Rhodes' work, the replacement of certain keywords by identifiers is done at a reduced cost. We also cut off the forward move prematurely, and assign a higher cost, if a reduction prompted by the change yields another error.

If none of the repairs considered so far survives the

† Thus in inserting the keyword `if` in our example, the method did not have enough lookahead to see the keyword `then`. The presence of a legal expression here indicates that either an `if` or a `while` is missing; the recovery method chooses the former without complete information.

forward move with sufficiently low cost, and if the state in which the error recovery mechanism is invoked has only one shift action, then the shiftable token is inserted. Since insertion of that token was previously considered to be unsatisfactory, the error recovery mechanism is usually invoked again after a small number of parsing steps. The second time, a "unique token" insertion is not considered.†

This aspect of the recovery mechanism has two advantages. It suggests to the user that that token seems to be appropriate in the particular context, and it allows the recovery mechanism to insert more than one token in certain cases.

4.4. Semantic considerations

The method includes the use of semantic information in the error detection and recovery mechanism. Before a given reduction occurs, the parser action routines can check that certain semantic conditions are satisfied. For instance, our grammar has no typed identifiers as lexical tokens. Typically, an action routine for a reduction of an identifier will determine its declared type and check that an identifier of that type can precede the next input token. If the semantic conditions are not satisfied, the syntax error recovery takes hold. The recovery action of changing the semantics to satisfy the condition, for instance changing an identifier type, is assigned a cost and a forward move is attempted. The resulting recovery cost is then compared with the various other changes described in the previous section.

As an example, if a function identifier were followed by "[", the array subscript bracket, rather than by "(", both a type error and a change from "[" to "(" would be considered.

These semantic checks are also used during the forward move, thereby eliminating changes that are syntactically correct but are meaningless. Returning to line 7 of Figure 1, the fact that "a" is an array causes the insertion of "[" rather than "(" even if the subscript expressions are more than five tokens long.

Semantic checks can also be used in place of grammar restrictions, when the language restrictions described by the grammar cause many programming errors from which it is hard to recover using local information. For example, in Pascal, the declarations of labels, constants, types and variables must be in sections occurring in that order, each section headed by the keyword `label`, `const`, `type` or `var` respectively. We let the grammar generate any number of these sections in any order, and impose the restrictions semantically. Consequently, errors stemming from extra sections or sections out of order are always handled properly. We also allow type declarations, rather than just type identifiers, in the grammar for parameter lists and the results of functions, again catching such errors semantically.*

Care is needed in introducing such syntactic extensions to make sure that the proper restrictions are always imposed by the action routines. The more general syntax may also cause the recovery mechanism to consider changes which are outside the language syntax. This could lead to misleading diagnostics unless the extra productions are carefully chosen and the semantic checks are made during the error recovery process.

4.5. Paired errors

One of the problems in this sort of error recovery is knowing whether a second error encountered during the forward move is an indication that our choice of repair was poor and should be rejected, or is just another error in the input stream. This problem arises on line 6 of Figure 1. The replacement of the identifier *go* by the keyword `goto` is considered here, but is rejected because there is another error ("`is`" rather than "`:=`") within the forward move. The error recovery is unsure of the change to `goto` here because of this other error and chooses to make a safer second level change. If the second error were not present, the error recovery would be more confident of its repair of the first error and would fix it successfully.

It would be possible to take into account reductions in the forward move rather than just distance into the forward context in the evaluation of costs. With this strategy, the change of "go" to `goto` allows the completion of a statement and partial progress into the next statement. It is thus a better repair than our current weighting scheme suggests.

4.6. Second level recovery

If no single terminal (first level) repair is found satisfactory; that is, if the cost of each change is higher than a threshold value, then the error recovery retreats to a set of second level changes. In this case the stack is popped to a state which has a shift on **error** and the parser shifts over the **error** token. The semantic actions for the error productions call a routine which advances the input to a symbol which is a legal shift in that state or a symbol which is one of a set of "beacon" symbols in the language. This part of the error recovery is similar to that of the basic YACC.

Second level recovery actions are reminiscent of "panic mode" recovery but differ in that the skipping of subsequent input is terminated either by context information (the shift symbols) or by context independent "beacons". Panic mode normally has only a set of "beacons".‡

As an example, if an error is encountered in an expression following an `if`, the parser will skip ahead to, but not past, the keyword `then` while "panic mode" recovery would most likely skip to ";" or the keyword `end`

† Care is needed to avoid looping in this case. It suffices to insist on shifting over a real input symbol for each unique input symbol inserted. For all other changes, the method requires that a recovery action result in a shift over one true input token, so no looping is possible.

* Not only do such changes facilitate the error recovery, but they also tend to simplify the grammar.

‡ Earlier work by [James 72] attempted to use shift information for a refined panic mode, but had no meaningful strategy for popping the parse stack.

Second level recovery actions are actually very similar to the recovery mechanism in Wirth's recursive descent recovery scheme [Wi 76]. Like Wirth, we also have a set of major section keywords which are never skipped.

In Pascal, the main program and each procedure or function consists of a declaration section and a statement section. An important cause of poor error recovery in languages of this form is a mistaken switch from one such section to the other. We use the beacons to handle that situation.

The second level recovery routine records in which section the error production which signals the input advance is used. If the second level recovery occurs in the declaration section and the keyword `for`, `repeat`, `while`, `goto`, or `if` is read, the keyword `begin` is inserted before the input keyword. If the second level recovery occurs in the statement section and the keyword `label`, `type`, `const`, or `var` is read, the stack is popped until a state is reached in which that keyword is a legal shift token. In either case, an appropriate message is generated.

The enumeration of lookaheads in states with shifts or **error** guarantees that a second level recovery of a major nonterminal which can be derived from another major nonterminal will cause another error to be detected if the input symbol (after the second level recovery action) cannot follow in the larger context. For example, if **error** causes a second level recovery to *expression* default reductions stop if a well formed expression will not complete a *statement.* If in fact an error in an expression yields another, second level, *statement* recovery without shifting any further input, then it is likely that the two errors are, in fact, one. We thus suppress further diagnostics after a second level recovery until some input is successfully shifted.

4.7. The lexical analyzer and input routines

The lexical analyzer must handle lexical errors gracefully, passing enough information on to the syntactic recovery mechanism so that it can decide what to do. The recovery mechanism needs to locate the position in the source program at which a token appeared so as to be able to print a meaningful diagnostic. Furthermore, the input routines must have the capability for suppressing the listing during a forward move so that the listing can later be produced after any error message printed for the current line.

Illegal characters encountered during lexical analysis are passed to the syntax analyzer, except that a sequence of occurrences of the same illegal character is replaced by a single such character. The lexical analyzer generates no error message, since the syntax analyzer will do so. The normal error recovery mechanism is then used, allowing possible replacement or deletion of the illegal character as appropriate.

Our lexical analyzer recognizes forms of the language syntax used at other installations and in other character sets. It also knows that string and comment delimiters must be paired and that several different characters may be used to delimit strings and character constants: normally, # for portability to other systems, and " and incorrectly.

Buffering input during a read ahead to gather context for forward moves is a nuisance but a necessity. Random access file i/o allowing only pointers to skipped lines to be saved makes listing suppression simple. If such access is not available, the text of lines can be spooled in core or to the disk.

4.8. Error messages.

The format of error messages produced by recovery systems has received too little attention. It is important that each error point be clearly marked, and very important that the messages produced be concise. Our method produces a single short message for each error. These messages are not wordy; they state simply the recovery action taken, and the symbol involved. It is important that users do not have to read through several lines of diagnostics to get the essential information being presented by the recovery method. This is especially true since most users of our system receive error messages at a time sharing terminal with a small (24 line) screen.

In many compilers, errors in declaring or defining an identifier in a program cause a plethora of diagnostic messages about identifier use, many of which stem from the same error and some of which may be misleading. We attempt to alleviate this problem in two ways.

When a misuse of an identifier such as inconsistent type or lack of declaration is detected by the semantic routines, a message is printed and the error and line number are recorded internally. Each time the same problem recurs, the line number of its occurrence is also recorded, but no additional inter−line message is generated. At the end of the program, procedure, or function, these errors are summarized.

If the second level recovery mechanism advances over an identifier, all subsequent warnings about identifiers, such as lack of initialization or lack of use, are suppressed.†

5. Cost of the implementation

5.1. Size of the implementation

Our grammar for Pascal consists of 184 rules, 16 of which are error productions. There are 69 terminals and 61 nonterminals. The resulting parser has 337 states of which 24 are states with shifts on error. The parse tables occupy 6000 bytes on a PDP−11. Of the 1700 entries in the parse action table, only 214 are lookaheads in states which have shifts on error, and some of these lookaheads would be present even if the error productions were not.

The total cost in space of the expansion of lookaheads in shifts on the `error` token is less that ten percent of the total table size. The version of YACC we worked with used a list representation of the LR parse tables. A newer version of YACC uses a matrix representation, and with this improved representation the increase is less than five percent.

† It would suffice to suppress warnings for the skipped identifiers only.

The recovery code itself requires little data beyond the basic parse tables. The additional fixed data required is the information as to symbol costs, which in any case is no more than a few hundred bytes, and the small tables which guide the second level recovery. The total size of our syntax analysis and recovery routines, for a full Pascal translator with a number of bells and whistles, is 17.5K bytes on a PDP−11. This breaks down roughly as follows:

Lexical analyzer	650 bytes	9%
Parser	800 bytes	5%
Input-Output	1700 bytes	10%
Error recovery	4800 bytes	27%
Parse tables	6000 bytes	34%
Action hooks	2500 bytes	15%

The input/output is an interface which provides spooling of input during the forward move, including file switching due to the placement of source code in multiple, possibly nested files. The action hooks are the semantic routine calls embedded in the grammar itself.

Without any error recovery, roughly 10K bytes of program and data would remain. In our environment, most of the additional size of the compiler is shared among users when multiple translations are taking place simultaneously, and it is thus less significant. The total size of the translator is roughly 70K bytes with 55K program shared among all users and 15K bytes unshared.

The method uses no additional space dynamically, and does not require copying of the parse stack. The forward move parser knows how to run on a "split-stack" so as to avoid the extra time and space involved in copying the stack.

5.2. Speed of the implementation

The translation time of programs without errors is not degraded by the method if a matrix form of the parse tables is used. In our list form we at first experienced a 15% reduction in speed due to the enumeration of lookaheads, and the subsequent slower linear searches of the list form parse tables in the expression chain. This 15% was recovered by more careful coding of the search loop in the parser.

The execution time of the error recovery itself is acceptably small. On one of the example programs from Rhodes' dissertation [Rhodes 73], a program of around 100 lines with almost 100 errors, the method used approximately 3 seconds of user processor time. A program of equivalent length with no errors would be processed in approximately 1 second of processor time. Since the error messages generated by the compiler are kept on secondary storage and must be retrieved to print each error message, a large percentage of this additional time was used just to retrieve the 100 or more error messages printed by the translator. The efficiency of the error recovery method has never proved to be a problem on our small systems running 20 to 30 simultaneous Pascal users.

6. Performance of the method

6.1. Quality of the recovery

We have gathered a large number of student programs over the course of an earlier quarter of instruction. The programs suggest that the method does indeed do a good job of correcting a large portion of programmer mistakes. The kind of errors which it does not repair often involve a deeper misunderstanding of the language than could be repaired by any known method with efficiency acceptable in our environment.

The error recovery also tends not to flurry, rarely producing a large number of diagnostics for a single syntax error. The reasons are a combination of the context checks for first-level repairs, the cost criteria, the summarized information, and the suppression of consecutive second level diagnostics.

6.2. A sample data set

We have obtained from Ripley and Druseikis [RD 78] a database of errors from erroneous student Pascal programs. There are 126 fragmentary programs in the sample set, and each program fragment represents a "unique" error in the original sample (which was much larger). The constitution of the larger sample is indicated by a weight for each fragment reflecting the frequency of its occurrence in the sample as a whole.

The samples were prepared for systems which do only syntactic analysis. The errors which the samples are intended to test are indicated, but declarations for identifiers used in the fragments are almost always omitted. Before running the samples we added declarations for the variables. This change allows our method to use semantic information which was present in the original programs, and treats the kinds of errors which were present there.

6.3. Evaluation of our method

We have made a preliminary evaluation of the performance of our algorithm on this data set using the criteria of the original paper [RD 78]. The paper classified error messages from the 6000-3.4 Pascal system as being either "accurate", "incorrect" or "poor". Messages were deemed to be poor if they were "vague, giving no indication of what was expected"[RD 78].

Our translator accurately diagnosed well over 80 percent of the errors in the weighted sample set, while Pascal 6000−3.4 accurately diagnosed roughly 50 percent of the errors. Many of the errors for which we give accurate repairs and Pascal 6000-3.4 does not involve ";" errors and incorrect keywords. The keyword errors include misspellings, omissions, and extra keywords. We also diagnose extra or out-of-order declaration or definition sections properly whereas the 6000-3.4 compiler does not.

Every example on which we rated our recovery as poor was rated poor by Ripley and Druseikis with one exception. In that example, the program contained

```
if y[n; = y[m] then . . .
```

Ripley and Druseikis rated the Pascal 6000-3.4 performance accurate because the message " ']' expected " was generated. However since the next three messages at the same point in the program were "illegal type of expression", "then expected", and "illegal symbol", we regard the diagnosis in this example as poor.

The same sample data set has been used by Pennello and DeRemer [PDeR 78] in evaluating their LR recovery algorithm and by Pai and Kieburtz [PK 79] in assessing their LL(1) strategy. However, they have used different evaluation criteria. We are in the process of evaluating our results using their criteria; the performance of our system again appears to be very good.

7. Conclusions and suggestions for further work

We have described an LR error recovery system embedded in a heavily used translator which provides high quality recovery with reasonable cost in space and time. The underlying mechanism is readily incorporated in parser constructors and the additional information is easily supplied by the implementer. If space is even more limited, there are natural places to prune the system. (For example, one could restrict the forward move to the number of tokens on the input line, provided there were at least one, and provided the cost formula were adjusted appropriately.)

The approach here is fundamentally limited. The limitation to single terminal repairs, which has made the method so efficient, may be difficult to remove without using much more space and time. This method does, however, provide a benchmark for more ambitious schemes. A method which proposes to use much more time than ours must provide substantially better error recovery to justify its cost.

There is much hope for automation of this method. Work is in progress to put this method in place within YACC so that it can easily be used. We are planning to continue work on error recovery, and are especially encouraged by the use of semantic information in this method. We intend to give more and careful study to the use of semantics in error recovery.

Acknowledgements

We are very grateful to Ken Thompson, who wrote the first version of the Berkeley Pascal System and provided a solid foundation on which to build. We thank David Ripley, Fred Druseikis, Ajit Pai, Dick Kieburtz, Tom Pennello, and Frank DeRemer, who shared the listings of their test results with us, so that we could attempt to make meaningful comparisons. Our thanks also go to Doug McIlroy, who made very helpful comments on the manuscript that significantly improved the presentation.

References

[AU 77]
Aho, A.V. and Ullman, J.D. *Principles of Compiler Design.* Addison Wesley, 1977.

[DR 76]
Druseikis, F.C. and Ripley, G.D. Error Recovery for Simple LR(k) Parsers. *Proceedings of the Annual Conference of the ACM* October, 1976.

[GR 75]
Graham, S.L., and Rhodes, S.P. Practical Syntactic Error Recovery. *Comm. ACM 16,* 11 (Nov. 1975), 639-650.

[James 72]
James, L. R. A Syntax Directed Error Recovery Method. M. S. Thesis, Tech. Report CSRG-13. Computer Systems Research Group, University of Toronto, May 1972.

[JW 75]
Jensen, K., and Wirth, N. *Pascal User Manual and Report.* Springer-Verlag, 1974.

[Johnson 77]
Johnson, S.C. YACC — Yet Another Compiler Compiler. Bell Laboratories, Murray Hill, 1977.

[JGH 79]
Joy, W.N., Graham, S.L., and Haley, C.B. *Berkeley Pascal User's Manual Version 1.1.* Computer Science Division, University of California at Berkeley, April 1979.

[MM 78]
Mickunas, M.D., and Modry, J.A. Automatic Error Recovery for LR Parsers. *Comm. ACM 21,* 6 (June 1978), 459-465.

[Modry 76]
Modry, J.A. Syntactic Error Recovery for LR Parsers. M.S. Thesis, Univ. Illinois, 1976.

[PDeR 78]
Pennello, T.J. and DeRemer, F.A. A Forward Move for LR Error Recovery. *Conf. Record ACM Symposium on Principles of Prog. Lang.,* January, 1978.

[PK 79]
Pai,A. and Kieburtz, R.B. Global Context Recovery: a New Strategy. *ACM Sigplan Symposium on Compiler Construction,* August, 1979.

[Rhodes 73]
Rhodes, S. P. Practical Syntactic Error Recovery for Programming Languages. Ph. D. Dissertation. Tech. Report #15. Computer Science Division, University of California, Berkeley. June, 1973.

[RD 78]
Ripley, G. David and Druseikis, Frederick C. A Statistical Analysis of Syntax Errors. *Journal of Computer Languages,*3, 1978.

[Wi 76]
Wirth, N. *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976.