An Assessment of Locally Least-Cost Error Recovery

Citation for published version:
Anderson, SO, Backhouse, RC, Bugge, EH & Stirling, CP 1983, 'An Assessment of Locally Least-Cost Error Recovery' The Computer Journal, vol. 26, no. 1, pp. 15-24. DOI: 10.1093/comjnl/26.1.15

Digital Object Identifier (DOI):
10.1093/comjnl/26.1.15

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Publisher's PDF, also known as Version of record

Published In:
The Computer Journal

Publisher Rights Statement:
Open Access Document

An Assessment of Locally Least-Cost Error Recovery*

S. O. Anderson, R. C. Backhouse,† E. H. Bugge and C. P. Stirling
Department of Computer Science, Heriot-Watt University, Edinburgh, UK

Locally least-cost error recovery is a technique for recovering from syntax errors by editing the input string at the point of error detection. An informal description of a parser generator which implements the technique is given. The generator takes as input an extended BNF description of a language together with a set of primitive edit costs and outputs a recursive descent syntax analyser including error recovery. Criteria for assessment of the technique are offered. Using these criteria the technique is assessed with respect to a database of over 100 example programs, and compared with an alternative local error recovery technique, that of follow set error recovery. The conclusion is that locally least-cost error recovery is more effective than follow set error recovery but much less economical in its use of storage space. The least-cost parser also runs between 15 and 20% slower than the follow set parser.

1. INTRODUCTION

Locally least-cost error recovery is a technique for recovering from syntax errors by editing the input string at the point of error detection. It was developed from earlier theoretical work on syntactic error correction[1–3] using an idea inspired by Fischer, Milton and Quiring.[4] The theory on which the method is based is described by Backhouse (Ref. 5, Chapt. 6) where the method is applied to the 'toy' programming language PLO invented by Wirth.[6]

Locally least-cost error recovery is essentially table-driven and not amenable to hand-encoding. But, by the end of 1980, we had completed the development of a parser generator which implements the technique. The generator inputs an extended BNF description of a language together with a set of primitive edit costs and outputs a recursive descent syntax analyser including locally least-cost error recovery and associated error messages. The purpose of this paper is to assess in some detail the usefulness of this tool and the effectiveness of locally least-cost error recovery.

A number of parser generators, automatically generate error recovery. Lewi *et al.*[7] employ a notion of synchro-triples to add error recovery to a recursive descent syntax analyser which, like ours, is generated from an extended BNF grammar. Pai and Kieburtz[8] use fiducial symbols to direct the error recovery in a table-driven parser generated from an LL(1) grammar. Error repair as the basis for error recovery has been used by Feyock and Lazarus[9] and Röhrich;[10] the idea of edit costs to choose among possible repairs goes back to Graham and Rhodes.[11] Closest to our own is the work of Fischer, Milton and Quiring[4] whose notion of insertion-only error correction is generalized by our notion of locally least-cost repair.

Few assessments of error recovery schemes have been given in any detail. Some evaluate the performance of their schemes on one short program; others just provide one or two small examples of the error recovery achieved. Our assessment is based on comparing the performance of locally least-cost error recovery with follow set error recovery[6,12,13] over 126 Pascal programs collected by Ripley and Druseikis.[14]

Section 2 of this paper gives an overview of the principal elements of locally least-cost error recovery and Section 3 describes the parser generator. Sections 4 and 5 discuss the basis for our assessment of the technique and Section 6 discusses its effectiveness in producing good error recovery and its efficiency with respect to storage and time overheads. Section 7 summarizes our results and examines the remaining weaknesses of the technique.

2. LOCALLY LEAST-COST ERROR RECOVERY

The principal idea governing locally least-cost error recovery is that the input symbol at an error-detection point may always be edited to some string which allows the parse to continue normally. The choice of string to which the input symbol is to be edited is effected by assigning costs to the primitive edit operations of inserting or deleting a single symbol, or changing one symbol to another. For example, suppose the parser encounters the following assignment within a Pascal program. (The ellipsis dots indicate unknown symbols following the identifier *c*.)

$$a := 2\,c\,\ldots$$

An error is detected on reading the input symbol *c*; it may be repaired by editing *c* to the empty word—effectively deleting *c*—by editing *c* to one of '+*c*', '**c*', '−*c*' etc.—effectively inserting an arithmetic operator—or by editing *c* to ';*c*'—effectively inserting a semicolon. Which string is chosen depends on the relative costs, assigned by the compiler writer, of deleting *c*, inserting

* This work was supported by grants from the UK Science and Engineering Research Council and the Carnegie Trust for the Universities of Scotland.

† Present address: Dept of Computer Science, University of Essex, Wivenhoe Park, Colchester, CO4 3SQ, UK.

CCC–0010–4620/83/0026–0015 $05.00

an arithmetic operator or inserting a semicolon. Supposing the latter is given the least cost the generated parser will produce the output

```
a := 2 c ...
         ^ ; inserted
```

and continue parsing as if the input had been '$a := 2;c$ . . .'.

A more substantial example of locally least-cost error recovery is shown in Example 1. The example is of a MODULA program written by one of the authors (Stuart Anderson). At the time he was a novice MODULA programmer and, inadvertently, he used **exit**, a reserved word, as the name of a procedure. Throughout the program **exit** has therefore been deleted. Thus, for example, the suggested repair of line 11 is

procedure *who*: **integer;**

```
1      interface module forkanddoor;
2      define enter, pickup, putdown, exit;
***                                       ^ symbol deleted
***                                        ^ identifier inserted
3      var forkinuse   : array 0:1 of boolean;
4          noinroom    : integer;
5          inroom      : array 0:1 of boolean
6          down0, down1 : signal;
***             ^ semicolon inserted
7      procedure enter(who : integer);
8      begin
9        noinroom := noinroom + 1; inroom[who] := true;
10     end enter;
11     procedure exit(who : integer);
***                   ^ symbol deleted
***                    ^
***                                  ^ symbol deleted
12     begin
13       noinroom := noinroom - 1; inroom[who] := false;
14     end exit
***           ^ symbol deleted
15     procedure pickup(fork : integer);
***              ^      ^ name misspelt
***                      ^ symbol deleted
***                          ^ ^      ^^
16     begin
17       case fork of
...
```

Example 1. Error recovery for MODULA.

Note that when the chosen repair consists of deleting a sequence of symbols the message 'symbol deleted' appears only under the first, the remaining symbols being marked simply by '^'. Note also that an insertion is indicated beneath the symbol *following* the place of insertion. Thus the message 'insert semicolon' beneath *down0* in line 6 indicates that a semicolon should be inserted immediately before *down0*.

Unlike many other proposed recovery schemes, locally least-cost error recovery is totally independent of any parsing algorithm. The only requirement on the parser is that it have the valid prefix property; it should announce an error as soon as a prefix of the input has been read for which there is no valid continuation. The LL(1), LR(1) and Earley's parsing algorithms all enjoy this property, and techniques for incorporating locally least-cost error recovery in them have been described elsewhere.[5, 15, 16]

Two deficiencies of locally least-cost error recovery are worth stressing. The first, the local nature of the choice of error repair, is well illustrated by our earlier example. If the actual input were

$$a := 2\,c := \cdots$$

then inserting a semicolon is clearly a plausible repair. However, had the input been

$$a := 2\,c + \cdots$$

its plausibility is much reduced. The second deficiency is the fact that repairs are made to the input symbol at the point of error detection. Thus the input

if $a = b$ *then* $x := 0$; **else** . . .

would be repaired by deleting the symbol **else** rather than the semicolon because it is not until the symbol **else** has been seen that an error in the input is detected.

Other deficiencies of the scheme are its inability to recognize multiple occurrences of the same error (for example, a misspelt identifier within a declaration will lead to 'undeclared identifier' errors wherever it is used) and to recognize systematic misuse of a language construct. An example of the latter would be the use of the Algol 60 **step** . . . **until** within a Pascal **for** statement. It also makes no attempt to handle semantic or context sensitive errors in the input.

The virtue of the scheme is that it makes no assumptions about the most likely kinds of errors—its only assumption is that any input string can be edited to a syntactically correct string by a sequence of insertions, deletions and changes—and yet it provides within the primitive edit costs a simple mechanism whereby the compiler designer may tune the error recovery to foreseen error patterns. Thus, at one extreme, the compiler designer may set all edit costs to infinity, in which case there will be no recovery—the compiler will abort whenever an error is detected. One may set all delete costs to finite values and all insert/change costs to infinity, in which case 'panic mode' error recovery will be invoked on error detection, causing whole sections of code to be skipped until an expected symbol is encountered. Or one may, as we have done, begin with a preconceived idea of the most likely errors (e.g. missing semicolons), assign low costs to the corresponding edit operations and then proceed to experiment with the costs, using a database of observed errors to adjust the edit costs until the recovery is the best possible within the limitations of the scheme. In summary, locally least-cost error recovery consists of the single mechanism of editing the next input symbol at least cost; but it incorporates flexibility by allowing edit costs to be chosen at will.

3. ELEMENTS OF THE PARSER GENERATOR

The software we have written generates a recursive descent syntax analyser, written in Pascal, incorporating locally least-cost error recovery. The input is an extended BNF description of the language[17] together with edit cost information for each terminal symbol. The bulk of our software is written in Pascal but we also make extensive use of facilities in Unix* primarily to overcome limitations of our Pascal compiler.

* Unix is a trademark of Bell Laboratories.

Using a PDP 11/34 we have successfully generated syntax analysers for Algol 68, MODULA and Pascal as well as for many toy programming languages. However, for real programming languages the software runs at the limits of the storage capacity of our machine; it would not be possible, for example, for us to generate an Ada syntax analyser.

There are four phases in the generation of a syntax analyser, which are as follows:

(1) *Checking phase.* In this phase three checks are carried out on the input grammar, namely a syntax check, a check for useless productions, and an LL(1) test. Any adverse results reported by the first two checks cause subsequent processing to be discontinued. On the other hand, if the grammar is not LL(1) a syntax analyser will still be generated but locally least-cost error recovery cannot always be guaranteed and the generated parser may require modifications to eliminate non-deterministic choices at the non-LL(1) positions.

(2) *Terminal symbol classification.* Two terminal symbols t and t' are in the same class when:

  (i) whenever t is a valid continuation of some input string then t' is so also, and vice versa

  (ii) t and t' have identical delete costs and the cost of changing t to any t'' is the same as changing t' to t''.

Classifying terminal symbols leads to considerable compression in the size of the sets and tables needed to determine error recovery. However, it does add some complexity to the parser since error recovery is defined with respect to the classes while correct parsing is done with respect to the symbols. For more details of the classification algorithm see Ref. 18.

(3) *Recovery policy determination.*

(4) *Final parser generation.*

The core of the system is phase 3 whose task is to construct a *table of recovery policies* for each production in the extended BNF grammar. These tables indicate, for every possible combination of input symbol and state of the syntax analyser, what action should be taken to simulate editing the input symbol. The general form taken by these tables is illustrated by Fig. 1, which contains the recovery policies for the two productions

$$st \rightarrow \textbf{if}\ u\ \textbf{then}\ \{sl\}\ [e]\ \textbf{fi}$$

$$sl \rightarrow st;$$

in a simple grammar.

Consider the first of these tables. The first row is the production and the first column is a list of the terminal symbols in the grammar. The remaining non-numeric entries are the recovery policies for each combination of production position and input symbol. Suppose, for example, that the recursive descent procedure *st* is called with input **if fi**. Then, according to the table, the symbol **if** is ok. The symbol **fi** following it is not; from the row labelled by **fi** in the upper table we see that to recover from the error one must ins(ert) the sequence of symbols *u* **then** {*sl*} [*e*] before recovery is achieved. Note that inserting an iterated construct like {*sl*} or an optional construct like [*e*] costs nothing because each may be empty.

st:	**if**	*u*	**then**	{*sl*}	[*e*]	**fi**
then	**	ins	ok	del	**	**
fi	**	ins	ins	ins 0	ins	ok
else	**	del	del	ins 0	call r+1	**
•	**	del	del	del r+1	** r+1	**
if	ok	ins	ins	call	**	**
;	**	ins	ins	call	**	**
id	**	call	cha	call	**	**
EOF	**	abort	abort	abort	**	**

sl:	[*st*]	;
then	**	del
fi	**	del 4
else	** r+1	del 1
•	** r+1	del 1
if	call	ins
;	ins	ok
id	call	ins
EOF	**	abort

Key

ok	input symbol and expected symbol agree
del	delete input symbol
ins	insert expected symbol
call	call procedure corresponding to expected symbol
cha	change input symbol to expected symbol
abort	no repair possible with given costs—abort parse

Numbers following a policy (e.g. del 4) indicate that the policy depends on the value of a parameter passed to the lhs procedure. Numbers beneath a policy indicate how these parameters are to be evaluated. Entries marked ** will never be accessed.

Figure 1. Tables of recovery policies.

Sometimes the recovery policy is contingent upon the history of the parse. For example, suppose one is parsing an Algol program and an error is detected on reading the symbol **else**. Two situations in which this might occur are shown below.

$$\textbf{if}\ a = b\ \textbf{then}\ x := 2*(y + z\ \textbf{else} \ldots \qquad (1)$$

$$\textbf{if}\ a = b\ \textbf{then}\ x := 0;\ \textbf{else} \ldots \qquad (2)$$

In (1) **else** is expected following a closing parenthesis. In other words the appropriate edit sequence is to insert ')' and then **else** is ok. In case (2) the semicolon terminates the conditional statement. Thus **else** is left dangling with no associated **if**-part. If backtracking is disallowed, as it is in our scheme, then the appropriate method of recovery is to delete **else** and resume parsing at the following symbol.

Such contingencies are handled by parameterizing some of the entries in the table of recovery policies. The second table in Fig. 1 shows three examples. Specifically, when a semicolon is expected and **fi**, **else** or '.' is encountered the chosen edit sequence is dependent on the value of a parameter passed to the procedure *sl*. This is indicated by the integer following each of the relevant entries in Fig. 1. Such integers are called *boundary costs* because they represent the boundary between choosing the tabulated policy or returning from the procedure call. The tabulated policy is called the *non-return policy*; the alternative, chosen when the parameter exceeds the boundary cost, is to continually ins(ert) symbols until the procedure has been left.

The entries beneath the policies in the upper table of Fig. 1 specify how the parameters are to be evaluated. We shall not discuss the parameter *evaluation mechanism* here since it is not relevant to our assessment. The interested reader is referred to Ref. 5. What is relevant to our assessment, however, is that the technique requires the storage of a potentially very large number of integer values. We have, therefore, expended a great deal of effort to develop automated techniques for reducing the size of the parameter and boundary cost tables. Classifying the terminal symbols[18] cuts down their size by about 40%, after which a global flow analysis of the parameters[19] makes significant economies, reducing the incidence of parameters and boundary costs to the level typified by Fig. 1. Statistics on the size of these tables for Pascal are given in Section 6.

Having computed the recovery policies the final phase is to encode them into a recursive descent syntax analyser. The result of encoding the tables in Fig. 1 is shown in Fig. 2.*

An important observation to be made about Fig. 2 is how closely the generated code resembles the code that a programmer might well write. This, we believe, is one of the most commendable features of our software since, compared to a table-driven parser, it will be relatively straightforward for an experienced compiler writer to amend or augment the basic syntax analyser.

The procedures *insert*, *change*, *delete* and *abortparse*, referred to in Fig. 2, perform very simple functions. *Insert* and *change* simply output an error message ('*x* inserted' or 'changed to *y*', respectively). *Delete* consists of a simple loop which outputs 'symbol deleted' and advances the input on each iteration and terminates when the input symbol is not in a given set.

Two types of set are evident in Fig. 2. The first consists of elements of type symbol, e.g. [*idsy*, ;*sy*, *fisy*], and the second elements of type class, e.g. [.*cl*, *thencl*]. The class sets are obtained by grouping together common elements in each column of a table of recovery policies (e.g. all elements for which the policy is *del*); the symbol sets are a by-product of the LL(1) test.

The above-mentioned process of parameterizing the error recovery is hidden away in five procedures. Three of these are shown in Fig. 2—*addparam*, *removeparam* and *pdelete*. *Addparam* and *removeparam* maintain a stack of parameter-table indices as the parsing proceeds. *Pdelete* is a more complicated version of *delete* which makes use of the parameter stack.

Significant modularity of the code generation routines in Phase 4 is achieved by giving a unique name to each production position in the grammar; each type of set is identified by a unique symbol which is prefixed to the position name. For instance the position names in Fig. 2 are *st*_101, *st*_102 etc., *d* is the prefix for a delete set, *a* is the prefix for an abort set, and so on. In this way the recursive descent routines can be generated separately from the set assignments. A disadvantage is that a very large number of scalar variables are generated, but this is rectified by using the C preprocessor provided with Unix to eliminate the majority of them before the parser

```
lst_0    := [idsy];
fsl      := [idsy, ;sy, ifsy];
fu       := lst_0;
fe       := [elsesy];
as_0     := [EOFcl];
dst_101 := [·cl, elsecl];
ast_101 := as_0;
dst_102 := dst_101;
gst_102 := [idcl];
ast_102 := as_0;
dst_103 := [·cl, thencl];
ast_103 := as_0;

procedure st (fstparamindex, lastparamindex : paramindex);
begin
addparam (fstparamindex, lastparamindex);
. . .
else begin
     next {skip if};
     delete(dst_101);
     if (cursymbol in fu)
     then u
     else if curclass in ast_101 then abortparse
     else insert (iu);
     delete (dst_102);
     if (cursymbol = thensy) or (curclass in gst_102)
     then begin
          if cursymbol <> thensy then change (gthen);
          next
          end
     else if curclass in ast_102 then abortparse
     else insert (ithen);
     delete (dst_103);
     while (cursymbol in fsl) do
        begin
        sl (mst_103, nst_103);
        delete (dst_103);
        end;
     if curclass in ast_103 then abortparse;
     if (cursymbol in fe)
     then e(mst_104, nst_104);
     next {skip fi};
     end;
removeparam(fstparamindex, lastparamindex);
end; {st}
procedure sl (fstparamindex, lastparamindex: paramindex);
begin
addparam (fstparamindex, lastparamindex);
if (cursymbol in fst)
then st (msl_0, nsl_0);
pdelete (dsl_1, esl_1, bsl_1);
if (cursymbol = ; sy)
then next
else if curclass in asl_1 then abortparse
else insert (i;);
removeparam (fstparamindex, lastparamindex);
end; {sl}
```

Figure 2. Code generated from policies in Fig. 1.

is compiled. Another disadvantage, which is less easy to rectify, is that we are obliged to use very short names for non-terminals—'*st*' instead of '*statement*', '*sl*' instead of '*statement_list*'. This is because both the C preprocessor and the Pascal compiler we use assume that all identifiers are unique up to the first eight characters. We get around this by using the Unix editor to abbreviate non-terminal names in the input grammar and then to reverse the abbreviations in chosen circumstances after all the processing is complete. The final effect is a highly readable, well-structured Pascal program with meaningful procedure names like *statement*, *expression*, etc.

A straightforward examination of the generated code enables one to make a number of preliminary remarks on its efficiency. First, compared to a syntax analyser with no error recovery, the overheads on parsing correct input

* The code in Fig. 2 is, of course, illegal in Pascal. The requirement imposed on the user in order to generate valid code is that all terminal and non-terminal symbols of the input grammar are valid Pascal identifiers. This requirement has been ignored here for greater clarity.

strings are: (1) evaluation of the parameters; (2) calls to *delete* or *pdelete*; (3) classification of the input symbols. Additionally, excluding calls to *next* (the lexical analyser), parsing incorrect strings is virtually equivalent to parsing the edited input suggested by the syntax analyser. In other words, there is a high correlation between the time-efficiency of the syntax analyser and its effectiveness.

4. BASIS FOR THE ASSESSMENT

Making an objective assessment of a recovery scheme developed by oneself is fraught with difficulties. Assessments of other schemes[20, 21] have used a simple classification of error diagnostics as 'good', 'acceptable' or 'poor'. But who is to judge the 'goodness' of a diagnostic? It ought not to be the implementor of the system, although that is who it invariably is. Other assessments, as noted by Ripley and Druseikis,[14] 'consist at best of an illustration of the technique on a few examples, almost invariably leaving the reader wondering how representative these examples are, and how well the technique works in general'.

In order to be as objective as possible we made two decisions. First, we have based our assessment almost entirely on a database of erroneous Pascal programs kindly supplied to us by G. David Ripley.[14] Second, we decided to compare our scheme directly with the 'follow set' scheme described by Wirth.[6] To this end two of our team (Edle Bugge and Colin Stirling) were given the tasks of producing the best possible error recovery for Pascal within the constraints of, respectively, locally least-cost error recovery and follow set error recovery. Both worked independently on the syntax analysers but collaborated on the assessment of the schemes; both were inexperienced in the schemes they were implementing when they began their tasks.

The systematics of follow set error recovery[6, 12, 13] is that with each state s of the parser there is associated a set $f(s)$ of *follow symbols*, those terminal symbols which are 'admissible' in state s. Should the symbol t returned by the scanner happen to be unacceptable in state s, then an error message is emitted and several symbols are skipped until a symbol t' in the set $f(s)$ is reached. The parser and the function f are tuned in such a way that the parser will advance to a state s' in which t' is accepted. Note that advancing to state s' is effectively equivalent to inserting some sequence of symbols which would in the normal course of events take s to s'. Skipping symbols is also equivalent to deleting them.

There is considerable variation in the choice of f. At one extreme only symbols acceptable in state s are in $f(s)$; this defines 'panic mode' error recovery in which symbols are skipped until an acceptable symbol is encountered. At the other extreme all terminal symbols are included in $f(s)$ giving 'insertion-only' error recovery.

Wirth[6] describes a systematic way of achieving a compromise between the two extremes but there is still latitude in his description. For instance, Wirth's code for recognizing {';' *statement*} (Ref. 6, p. 343) is

```
while sym in [semicolon] + statbegsy do
  begin
  if sym = semicolon then getsym else error(10);
  statement ([semicolon, end] + fsys)
  end
```

whereas his code for recognizing {',' *vardeclaration*} (Ref. 6, p. 345) is

```
while sym = comma do
  begin getsym; vardeclaration
  end
```

In both cases zero or more repetitions of a construct of the form tA are being recognized, where t is a terminal and A is a non-terminal symbol. By testing for both *sym* **in** *statbegsys* and *sym* = *semicolon* in the former, one is anticipating the possibility of a missing semicolon; in contrast, a missing comma is not anticipated in the latter segment of the code.

A second instance of the latitude in Wirth's scheme is in the recognition of terminal symbols. Typically the code may take the form (Ref. 13, p. 237)

```
if sy = semicolon then insymbol else error(14)
```

or (Ref. 13, p. 229)

```
if sy = semicolon then insymbol else
  begin error(14);
    if sy = comma then insymbol
  end
```

Both conditional statements accept semicolons but the second anticipates the use of a comma instead of a semicolon whereas the first does not. In terms of error repair, the second conditional simulates the operation of changing a comma to a semicolon.

Because of this latitude in Wirth's scheme (which, by the way, is not well documented) we have been obliged to experiment in order to achieve the best possible results with the scheme. The statistics we give for the assessment of the follow set scheme are therefore themselves the outcome of a separate inquiry into its effectiveness. For further details see Ref. 22.

There are many similarities between locally least-cost error recovery as implemented in a recursive descent syntax analyser and follow set error recovery. Both recover at the point of error detection without backtracking over the input, both use one-symbol lookahead to evaluate the recovery policy. Neither contains a mechanism for handling context-sensitive errors or repeated misuse of a language. Finally, both are intended to offer a compromise between economy and effectiveness; they should provide sensible diagnostics with only a marginal overhead on computational resources. The comparison we make is therefore one of like with like. It is made more valuable by the fact that follow set error recovery is undoubtedly popular and is also recommended for use in an environment where introductory programming is taught.[13]

Our assessment of the follow set scheme may be unorthodox and so requires explanation. It is common when using this scheme to simply indicate the symbol or symbols expected at the point of error detection. Thus, for example, Wirth's implementation of Pascal-S[13] includes a preponderance of messages like 'dot expected', 'then expected' and 'a constant cannot begin with the indicated symbol'. Sometimes the recovery action taken by the compiler is obvious from the error message—if 'semicolon expected' appears it is often safe to assume that the compiler has effectively inserted a semicolon at the error detection point—but, more often than not, the diagnostic whitewashes over the recovery action. Commonly one or more symbols may be skipped before parsing resumes but it is rare for this to be indicated to the user. To get a clearer and, we believe, more objective appraisal of follow set error recovery we have modified the diagnostics so that they indicate the effective repair of the input string made by the parser. Thus, if a symbol is skipped the diagnostic is 'symbol deleted'; if a particular symbol x is expected and effectively inserted (for example a semicolon) the diagnostic is 'x inserted'; finally, if a symbol x is expected and the input symbol is effectively changed to x (as might happen for instance when '=' instead of ':=' is used in an assignment) the diagnostic is 'changed to x'.

The criticism that may be made is that in so doing we have made follow set error recovery look like a scheme for error repair when it was never originally conceived as such. Our counter to this criticism is that we are attempting to assess the error *recovery* rather than the *diagnostics*. Good diagnostics are vital, of course, but their design is a separate issue from the design of the error recovery. Good diagnostics are useless, however, if the error recovery is poor!

Note that our comparison is of the 'bare' schemes, locally least-cost versus follow set error recovery, although for a truly practical compiler we would normally recommend augmenting both by the use of error productions[23] or semantic routines.[9] There are two exceptions to this statement. First, the strict syntax of Pascal insists on declarations in the order **label-**, **const-**, **type-**, **var-**, and **procedure/function-**declarations. Following common procedure our syntax analysers allow declarations in any order but code has been added to flag an error if the given order is violated. Second, Pascal is non-LL(1) in that an identifier may begin an assignment statement or a procedure call. We overcome this problem in different ways in the two analysers, by using a primitive symbol table in the locally least-cost analyser and by using a modified LL(1) grammar in the follow set analyser. For further details see Refs 22 and 24.

5. CRITERIA FOR ERROR REPAIR

Ripley and Druseikis[14] report that the most common type of Pascal programming error is a single symbol error. Hence, in actual practice, one can often speak of the 'most plausible' error repair. This is the central criterion for 'best repair' that we use. However, there are cases where there is a group of equally plausible repairs and other cases where there just is not a plausible repair. (An instance of the latter is where the error is a 'wrong language' error, for example 'f:= **if** $x = 0$ **then** 1 **else** $x*f(x-1)$' in a Pascal program.) In these cases the minimum distance criterion[1] is useful. The distance of a repair from the input string is the sum of the primitive edit operations used, where a single insert, delete or change counts as one. A minimum distance repair is one closest to the input string. Thus, we use this criterion to thin down an initial set of equally plausible repairs, and to constitute the definition of 'best repair' in those circumstances where there is not a unique most plausible repair. Note that, as in example 4 below, a minimum distance repair can be implausible.

An error recovery scheme is not only to be judged upon what repairs are made, but also upon how it actually makes those repairs. This accounts for the different categories (i) and (ii) below. We have divided the repairs made by a recovery scheme into seven groups.

(i) The recovery scheme repairs to the most plausible repair and it does so in minimum distance. (Abbreviated MPR)
(ii) The recovery scheme repairs to the most plausible repair but does not do so in minimum distance. (Abbreviated PR)
(iii) The recovery scheme repairs to a minimum distance repair which is not the most plausible repair. (Abbreviated MD)
(iv) The repair is not the most plausible (and sometimes not at all plausible) nor minimum distance but has no effect upon the parsing of subsequent symbols. (Abbreviated R)
(v) The recovery scheme repairs neither to the most plausible repair, nor to a minimum distance repair and does so by entering into panic mode. That is, it deletes a number of symbols until a 'safe' symbol is found. (Abbreviated P)
(vi) The recovery scheme cascades; that is, it repairs in such a way as to cause errors to be found in later correct parts of the program. (Abbreviated C)
(vii) The repair satisfies both (v) and (vi). (Abbreviated PC)

The difference between categories (i) and (ii) is illustrated by examples 2 and 3. The resulting repaired program is

```
1     program p(input, output); begin begin
2     end ·
***     ^ end inserted
```

Example 2. MPR repair.

```
1     program p(input, output); begin begin
2     end ·
***     ^ symbol deleted
***       ^ end inserted
***       ^ dot inserted
```

Example 3. PR repair.

```
1     program p(input, output);
2     var count, listdata : array[1..2] of integer;
3     begin
4       if count[listdata[sub] := 0
***                          ^ changed to plus
5       then begin f := listdata[sub];
***       ^ ] inserted
6     end; x := 1 end.
```

Example 4. MD repair.

```
1     program p(input, output);
2     function search(i: integer; x: order; ) : boolean
***                                          ^ id inserted
***                                          ^ colon inserted
***                                          ^ type id inserted
3     var q: integer; begin x := 1 end; begin end.
```

Example 5. R repair.

```
1     program p(input, output); begin
2       repeat writeln(' input is:', number)
3         if number> 1
***        ^ symbol deleted
***              ^^^
4         then x := 1 until x = 1 end.
***           ^^  ^  ^
```

Example 6. P repair.

the same in both cases and is the most plausible. But, unlike the former, the latter does not make the repair in minimum distance.

Example 4 is an instance of category (iii). The repair is minimum distance but is not a plausible repair.

Group (iv) consists of those repairs that are neither plausible nor minimum distance and also are not the result of panic or cascade. Example 5 is a case in point.

An instance of category (v) is given by example 6. Here the follow set scheme deletes until it reaches the 'safe' symbol **until**. This is an instance of a highly implausible repair caused by the error recovery scheme entering into panic mode. Such is a central facet of follow set error recovery, but in many cases it results in a most plausible repair (for instance, when a single symbol only is deleted). Hence, category (v) is intended to capture only those instances of implausible repairs which are a result of panic mode.

Sometimes an error recovery scheme repairs an error in such a way that it later detects errors in the program which are not errors. This 'cascading' of error messages happens in example 7. Example 8 is a mixture of cascading and panic mode.

```
1    program p(input, output); begin
2       writeln ('the call to getelement resulted in no
3                  nodes avail',);
***                             ^ symbol deleted
***                             ^
4       x := 1 end.
***        ^ changed to plus
***              ^ ) inserted
```

Example 7. C repair.

```
1    program p(input, output); begin
2       repeat writeln(' input is:', number)
3          if number > 1
***          ^ symbol deleted
***                     ^ until inserted
4          then x := 1 until x = 1 end.
***           ^ symbol deleted
***           ^ ^ ^    ^ ^ ^ ^
```

Example 8. PC repair.

We divide the seven groups into three larger groups as a means to offering an overview of the recovery schemes.

Good repair	—contains groups (i) and (ii)
Marginal repair	—contains groups (iii) and (iv)
Poor repair	—contains groups (v), (vi) and (vii)

Hence, a good repair is a most plausible repair or, in the case where there is not a plausible repair, a minimum distance repair. Marginal repairs, on the other hand, are not generally acceptable; they represent incorrectly diagnosed errors but have some virtues, unlike poor repairs which cause cascading or may result in other errors not being observed.

6. RESULTS AND EXPERIENCE WITH THE RECOVERY SCHEMES

6.1 Effectiveness

Each program from Ripley and Druseikis's database has been put into one of two main groups; those containing a single symbol error, and those containing two or more. Following standard practice we count as a single error program one that contains more than one instance of the same error.

Four of the 126 programs, those whose only error is a declaration out of order, do not appear in our results. This is because the grammar of Pascal we have used does not impose an ordering on the declarations, and so this is not picked up by the recovery schemes. The remaining 122 programs are distributed as follows.

single error programs	76
multiple error programs	46

Table 1 gives the overall results using the categories 'good', 'marginal', and 'poor'. Table 2 is a breakdown of these results using the categories we introduced in the last section. Note that the acronyms LLC and FS are used in these and the following tables for locally least-cost and follow set error recovery, respectively.

Table 1. Overall results for the 122 examples

	LLC	FS
Good	70	55
Marginal	26	33
Poor	26	34

Table 2. Breakdown of overall results into the 7 categories

	LLC	FS
MPR	64	54
PR	6	1
MD	6	8
R	20	25
P	4	18
C	20	24
PC	2	2

The conclusion from these results is that locally least-cost error recovery is better on aggregate than follow set error recovery, particularly in avoiding panic mode recovery. Note, though, that the difference in their ability to avoid cascading of error messages (group C in Table 2) is marginal. We now consider the results for each of the main groups in turn.

(a) Single error programs. The 76 single error programs have been subdivided as follows

(i)	The most plausible error repair is a single insert	34
(ii)	The most plausible error repair is a single delete	11
(iii)	The most plausible error repair is a single change	20
(iv)	The single error is a misuse or a misspelling of a keyword	11

The overall results of the single error programs are given in Tables 3 and 4. We have counted a program as belonging to the first of these subdivisions if the most plausible error repair may either be a single delete or a single insert.

Table 3. Results for the 76 single error examples

	LLC	FS
Good	49	40
Marginal	15	17
Poor	12	22

Table 4. Breakdown of single errors into the 7 categories

	LLC	FS
MPR	46	39
PR	3	1
MD	2	2
R	13	15
P	1	4
C	9	13
PC	2	2

Both schemes yield their best results on these errors. Locally least-cost error recovery offers almost 75% of 'good' repairs for each of the error types described above.

Four of the single-symbol errors in the sample involved erroneous string delimiters (wrong symbol or delimiter missing). Neither scheme was able to handle these well because such errors, and those in comment delimiters, are lexical errors in our parser. Strings are recognized by the lexical analyser and passed on to the syntax analyser as single symbols. On the other hand, comments are removed by the lexical analyser. Therefore these errors can never be repaired in the most plausible fashion.

There are a few errors in this group that cannot be repaired satisfactorily without some form of backtracking. Example 9 is a case in point. Here the most plausible repair is to replace **procedure** with **function**, but this error can not be detected at the point it occurs because the occurrence of **procedure** in this position is valid.

```
1     program p(input, output);
2     procedure factorial (x: integer; var fact: integer)
3                  : integer;
***             ^ changed to semicolon
***                    ^ symbol deleted
***                    ^
4     var q: integer; begin x := 1 end; begin end.
```

Example 9. Local error recovery is unsatisfactory.

The single insert repair results (Table 5) are quite good for both schemes, but better for locally least-cost error recovery. On one type of error—a missing semicolon, which is by far the most common single-symbol error made by Pascal programmers—the scheme has a 100% record of success.

Table 5. Number of 'good' repairs for each error type

	Total no. of programs	Number of good repairs	
		LLC	FS
Single insert	34	29	25
Single delete	11	7	6
Single change	20	11	8
Keyword error	11	2	1

Unlike the results for single symbol change, those for misuse/misspelling of keywords are not good (Table 5). This is, in part, because none of the schemes can correct spelling errors and a misspelt keyword will be regarded as an identifier. Local error recovery techniques are not particularly suitable for correcting this type of error. Example 10 is a case where the repair is very poor. Therefore, a simple spelling correction routine inside the parser would have improved the performance on this group, especially in cases like example 10.

```
1     program p(input, output); begin repeat
2       writeln (' ---------- ');
3       untill eof (input);
***             ^ := inserted
4       x := 1 end.
***                 ^ symbol deleted
***                ^
***                   ^ until inserted
***                   ^ id inserted
***                   ^ end inserted
***                   ^ dot inserted
```

Example 10. Keyword error.

(b) Multiple error programs. Multiple error programs are those programs where a most plausible repair involves more than one edit operation. We include in this group complementary errors—for example the use of parentheses ( ) instead of brackets [ ] when indexing an array. Also included are 'wrong-language' errors for which, as discussed earlier, we count the minimum distance repair as the most plausible.

Twenty-two of the programs in this group contained errors which could be repaired by just two edit operations and, of these, the errors occurred together in twelve cases.

Table 6. Overall results for the 22 two error programs

	LLC	FS
Good	12	9
Marginal	5	11
Poor	5	2

Table 7. Breakdown of the two error programs into the 7 categories

	LLC	FS
MPR	11	9
PR	1	0
MD	3	5
R	2	6
P	0	1
C	5	1
PC	0	0

Table 8. Overall results for the multiple error programs

	LLC	FS
Good	21	15
Marginal	11	16
Poor	14	15

Table 9. Breakdown of the multiple error programs

	LLC	FS
MPR	18	15
PR	3	0
MD	4	6
R	7	10
P	3	4
C	11	11
PC	0	0

In such circumstances both schemes did well (8 good repairs for locally least-cost and 7 for follow set error recovery). Their performance on the remaining errors was, as might have been expected, relatively poor. Follow set error recovery did particularly badly on the five examples of complementary errors.

6.2 Efficiency

Although locally least-cost error recovery may more effectively diagnose errors, it does so at the expense of economy in the use of storage space. This is evident from the difference in size of the object code of the two parsers we used in our assessment; the parser using the follow set scheme compiled into 19 146 bytes, the parser using the locally least-cost scheme compiled into 26 618 bytes of storage space.

These figures are not the whole picture, however, since a large amount of code in the locally least-cost parser consisted of assignments to set up the various tables and sets needed to drive the error recovery. Table 10 shows a breakdown of the two programs into their various components. The large difference in the number of lines of code is partially explained by the declaration and initialization of the class sets in the locally least-cost scheme. More significantly, though, the recursive descent routines are 50% longer when using least-cost recovery than when using follow set recovery. This is, perhaps, the best indicator of the extra complexity of the former scheme.

Table 10. Size of source code in lines

	LLC	FS
Lexical analyser	135	135
Recursive descent routines	1600	1035
Declarations	246	81
Error procedures	—	151
Initializations		
Symbol/class sets	516	162
Others	381	155
Total program length	2878	1719

Each class set declared in the least-cost parser is a bit vector of length 38 and so occupies six bytes. There are 140 class sets; the follow set scheme used 56 sets for a comparable purpose. The symbol sets are used by both parsers; there are 68 of them and each occupies eight bytes.

The figures in Table 10 do not include details of the boundary and parameter tables which had, respectively, 986 and 660 entries. Each entry in the boundary table occupies two bytes and in the parameter table three bytes. Our Pascal grammar has 69 terminal symbols and 368 production positions; thus, the worst-case size of these tables is 69 × 368 which is approximately 25 000 entries each. Indeed, the boundary table is larger than necessary because, for ease of access, it is stored as a two-dimensional array; only 228 elements in this array are actually defined. By storing the entries as a one-dimentional array together with an access table one could reduce the storage requirements considerably but at the expense of a small increase in access times.

We have timed both schemes over the complete database of programs and over themselves. For the least-cost scheme the most dominant factor in its runtime was the large number of initializations. This, however, is a limitation of Pascal rather than the recovery scheme—by using the unit facility in UCSD Pascal, for example, the initializations could all be performed at compile-time and so would have no effect upon the runtime. Excluding the time taken for performing initializations, the follow set parser took 293 s whereas the least-cost parser took 359 s to parse the complete database of programs, a difference of approximately 20%. When the parsers were used to parse the source codes of both schemes the least-cost scheme was approximately 15% slower.

7. CONCLUSION

This paper has described a parser generator which generates highly readable Pascal code from a given extended BNF grammar. The code includes locally least-cost error recovery which can be tuned to anticipated error patterns by setting primitive edit costs. There is, therefore, a great deal of flexibility in the standard of error recovery achieved. Meaningful error messages describing the repairs made by the parser are also automatically generated.

Locally least-cost error recovery, according to our assessment, is more effective than follow set error recovery. Overall, 57% of the repaired example programs are diagnosed as 'good' whereas the figure for the follow set scheme is only 45%. In more detail, the figures are 64% in contrast to 53% and 46% in contrast to 33% for programs containing single and multiple errors, respectively. This increase in effectiveness is paid for by a loss in efficiency, particularly with respect to the storage requirements. Moreover, the least-cost scheme still retains the same fundamental limitations as the follow set scheme; repairs are made at the point of error detection, multiple occurrences of the same error are not recognized and no mechanism exists for handling context-sensitive or semantic errors. Consequently, the improvement offered by the least-cost scheme over the follow-set scheme is not as significant as the statistics may suggest.

There are improvements which could be made to the least-cost technique which, we believe, would make it viable. These improvements are, first, a further reduction in the size of the parameter and boundary cost tables and, second, the development of a calculus for setting the primitive edit costs. At the present time work on these developments is in its first stages.

The main cause of the excessive size of the parameter and boundary cost tables is that they contain integer values; the follow sets, on the other hand, are essentially Boolean values. However, the parameters perform a Boolean function—they serve to indicate whether the cost of a set of edit operations is greater than a given boundary cost or not. In some circumstances, therefore, it is possible to map the parameters homomorphically to Boolean values and hence store them as bit vectors. In other circumstances their function is as counters, for example counting the depth of parenthesization of an expression, and sometimes the same parameter is compared against more than one boundary value. At present we have no algorithms to test whether a given parameter can be replaced by a Boolean and it remains to be seen whether the reduction in storage space so obtained would be worthwhile.

The possibility of developing a calculus for setting the costs has been, for us, a major motivation for the use of such costs. The idea is that one should be able to compute an optimal set of edit costs from a sample of error patterns and the preferred repairs for each of the error patterns. We ourselves have partial solutions to this problem.[24] Elsewhere, N. Kotarski (private communication) claims some success in applying integer programming techniques to the problem but much work needs yet to be done. Were such a calculus to be developed, however, it would add immensely to the convenience and flexibility of our system.

Acknowledgement

Thanks go to M. H. Williams for his careful reading and criticism of this paper.

REFERENCES

1. A. V. Aho and T. G. Peterson, A minimum distance error correction parser for context-free languages. *SIAM Journal of Computing* **1**, 305–312 (1972).
2. R. A. Wagner and M. J. Fischer, The string-to-string correction problem. *Journal of the ACM* **21**(1), 168–173 (1974).
3. R. A. Wagner, Order-*n* correction for regular languages. *Communications of the ACM* **17**(5), 269–271 (1974).
4. C. N. Fischer, D. R. Milton and S. B. Quiring, An efficient insertion-only error corrector for LL(1) parsers. *Acta Informatica* **13**, 141–154 (1980).
5. R. C. Backhouse, *Syntax of Programming Languages: Theory and Practice*, Prentice-Hall International, London (1979).
6. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, New Jersey (1976).
7. J. Lewi, K. DeVlaminck, J. Huens and M. Huybrechts, The ELL(1) parser generator and the error recovery mechanism. *Acta Informatica* **10**, 208–228 (1978).
8. A. B. Pai and R. Kieburtz, Global context recovery: a new strategy for syntactic error recovery by table-driven parsers. *ACM Transactions on Programming Languages and Systems* **2**(1), 18–41 (1980).
9. S. Feyock and P. Lazarus, Syntax-directed correction of syntax errors. *Software—Practice and Experience* **6**, 207–219 (1976).
10. J. Röhrich, Methods for the automatic construction of error correcting parsers. *Acta Informatica* **13**, 115–139 (1980).
11. S. L. Graham and S. P. Rhodes, Practical syntactic error recovery. *Communications of the ACM* **18**(11), 639–650 (1975).
12. U. Amman, The Zurich implementation. In *Pascal—The Language and its Implementation*, ed. by D. W. Barron, Wiley, Chichester (1981).
13. N. Wirth, Pascal-S: a subset and its implementation. In *Pascal—The Language and its Implementation*, ed. by D. W. Barron, Wiley, Chichester (1981).
14. G. D. Ripley and F. C. Druseikis, A statistical analysis of syntax errors. *Computer Languages* **3**, 227–240 (1978).
15. S. O. Anderson and R. C. Backhouse, Locally least-cost error recovery in Earley's algorithm. *ACM Transactions on Programming Languages and Systems* **3**(3), 318–347 (1981).
16. S. O. Anderson and R. C. Backhouse, Locally least-cost error recovery in LR parsers: a basis, submitted for publication.
17. N. Wirth, What can we do about the unnecessary diversity of notation for syntactic definitions. *Communications of the ACM* **20**(11), 822–823 (1977).
18. R. C. Backhouse, Classifying terminal symbols in LL(1) grammars, submitted for publication.
19. R. C. Backhouse, Two global data flow analysis problems arising in locally least-cost error recovery, submitted for publication.
20. K-C. Tai, Syntactic error correction in programming languages. *IEEE Transactions on Software Engineering* **SE-4**(5), 414–425 (1978).
21. T. J. Pennello and F. DeRemer, A forward move algorithm for LR error recovery. *Conference Record 5th ACM Symposium on Principles of Programming Languages,* 241–254 (1978).
22. C. P. Stirling, Follow set error recovery. *Research Report*, Heriot-Watt University, Department of Computer Science (1981).
23. C. N. Fischer and J. Mauney, On the role of error productions in syntactic error correction. *Computer Languages* **5**, 131–139 (1980).
24. E. Bugge, Implementing and assessing locally least-cost error recovery for Pascal. MSc. Thesis, Heriot-Watt University, Department of Computer Science (1982).

Received December 1981