A Practical Minimum Distance Method for Syntax Error Handling

J. A. Dain

Department of Computer Science
University of Warwick
Coventry, CV4 7AL

Summary

This paper presents a method for recovering from syntax errors encountered during parsing. The method provides a form of minimum distance repair, has linear time complexity, and is completely automatic. It is incorporated into the LR parser-generator *yacc* in such a way that a compiler writer can generate a parser with error recovery without providing any additional information to *yacc*. Error messages phrased in terms of source input are generated automatically. A formal method is presented for evaluating the performance of error recovery methods, based on global minimum-distance error correction. The minimum-distance error recovery method achieves a theoretically best performance on 80% of Pascal programs in the Ripley-Druseikis collection. Comparisons of performance with other error recovery methods are given.

Introduction

Language translators such as compilers and interpreters are an essential component of every software development system, and indeed every software system in which some sort of dialogue is conducted between user and computer, such as a spreadsheet or a database package. A language translator includes a syntax analyser which must make provision for errors in its input, otherwise the translator will not be robust. Classic recommendations for "what the compiler should tell the user" were made by Horning[1] in 1974. The problem of syntax error handling has received considerable attention over the last two decades, including various surveys.[2,3,4] Yet the "perfect" syntax error handling scheme has not appeared. Such a scheme would combine the ability to construct a perfect repair for incorrect input with the linear time complexity of efficient parsers for the deterministic context-free languages.

What would constitute a perfect repair? Minimum or Hamming distance is conventionally used as a formal model which approximates the user's concept of syntax errors, as it measures the *shortest* way to transform erroneous input into correct input.[2,5,6] Given a set of edit operations, typically insertion, deletion or replacement of a single symbol, the minimum distance between two strings is the minimum number of edit operations required to transform one string into the other. In the context of parsing, the target string is a sentence of the context-free language, into which the syntactically incorrect input string must be transformed. So a perfect repair is a sentence nearest to the actual input, in the sense that there is no sentence whose minimum distance from the input is smaller. Algorithms for global correction, which aim to construct such repairs, exist[5,7,8,9,10] but are typically based on Earley's algorithm[11] for general context-free parsing and are not practical, requiring $O(n^3)$ time and $O(n^2)$ space.

This paper presents a method for syntax error recovery which aims to provide a form of minimum distance repair, has linear time complexity, and can be incorporated into an LR parser-generator. The method is described in detail and it is shown how error messages can be generated automatically. Finally a method for performance evaluation based on minimum distance is presented and used to evaluate several error recovery schemes.

Minimum Distance Recovery

The recovery method presented here will be invoked by a parser for a context-free language (CFL) L at the point at which the parser detects an error in its input. It will repair the input, returning control to the parser in a configuration in which the remaining input can be parsed. An LL or LR parser possesses the correct prefix property, so when such a parser detects an error, the input parsed is a prefix of some sentence of the language, and hence there exists at least one suffix or *continuation string* whose concatenation to the parsed input forms a sentence of L. The idea behind the method is to generate a set of prefixes of continuation strings for the input already parsed, and to choose from it a string whose minimum distance from a prefix of the remaining input string is minimum over the set. This string is used to replace part of the input.

Definitions

Definitions are given for a sequence of edit operations from one string to another, and the resulting minimum distance measure, following Wagner and Fischer.[12] The concept of the parser-defined error in a string with respect to a CFL L, due to Peterson,[13] is useful for modelling the point at which a prefix of the string ceases to form a prefix of L. The set of continuation strings for any prefix of L is also defined.

Let Σ be a finite alphabet of symbols. a, b denote symbols in Σ, u, v denote strings over Σ, and ϵ denotes the empty string. Let Δ be the set of edit operations $\{(a, b) | a, b \in \Sigma \cup \{\epsilon\}, (a, b) \neq (\epsilon, \epsilon)\}$.

Definition 1 *For strings u, v in Σ^*, $u \to v$ via $a \to b$ if $(a, b) \in \Delta$ and there are strings w, x in Σ^* such that $u = wax$ and $v = wbx$.*

Definition 2 *$u \to v$ via E if E is a finite sequence of edit operations $e_1 \ldots e_n$ and there are strings $w_1, \ldots, w_{n-1}$ in Σ^* such that $u \to w_1$ via e_1, $w_i \to w_{i+1}$ via e_i for $i = 1, \ldots n-2$, and $w_{n-1} \to v$ via e_n.*

Definition 3 *The* minimum distance *$d(u, v)$ between two strings u, v is given by $d(u, v) = \min\{n \mid u \to v$ via E for some sequence of edit operations $E = e_1 \ldots e_n\}$.*

Definition 4 *The* set of prefixes *$Pre(L)$ of a language L over Σ^* is given by $Pre(L) = \{u \in \Sigma^* \mid uv \in L$ for some $v \in \Sigma^*\}$.*

Definition 5 *The* parser-defined error *for a string u with respect to a CFL L, where $u \notin L$, is after the string x at the symbol a where $u = xay$, $x \in Pre(L)$ and $xa \notin Pre(L)$.*

Definition 6 *The* set of continuation strings *$Cont(u)$ for a prefix u of a CFL L is given by $Cont(u) = \{w \in \Sigma^* \mid uw \in L\}$.*

Outline of the method

Our goal is to find a practical method for error recovery using minimum distance as a criterion for choosing a repair. The method will be invoked when the parser detects an error in the input, that is, for a correct-prefix parser, at the parser-defined error. Ideally the repair would be the continuation string whose minimum distance from the remaining input is minimum over the set of all continuation strings, but this is impractical. In order to devise a practical method, we place a limit on the number of possible repair strings by generating prefixes of continuation strings, these prefixes being of length up to a fixed number σ. The prefixes are measured against a fixed number τ of remaining input symbols, to find the best match. The chosen prefix is then used to replace a prefix of the remaining input. The method does not guarantee to find a repair leading to a sentence of the language, only that a number of the remaining input symbols will be accepted by the parser.

Two problems to be solved are which continuation string prefix to choose, and how much of the remaining input to replace. It is desirable to obtain the closest possible match between continuation string prefix and remaining input, but not necessarily by replacing all the input symbols used in the minimum distance measure. The closest match is obtained by choosing the continuation string with smallest minimum distance from some *prefix* of the fixed amount of input.

The minimum distance method can now be outlined as follows.

1. Let the input string be represented by uv where the parser-defined error is after u and $v = v_1 \dots v_n$, for $v_i \in \Sigma$, $i = 1, \dots, n$.
2. Let $R = \{w \mid uw \in L, |w| < \sigma\} \cup \{w \mid uw \in Pre(L), |w| = \sigma\}$, where σ is a fixed integer.
3. Choose x in R and m, $1 \leq m \leq \tau$, such that $d(v_1 \dots v_m, x) \leq d(v_1 \dots v_j, y)$ for all $j, 1 \leq j \leq \tau$, and all y in R, where τ is a fixed integer.
4. The repaired string is $uxv_{m+1} \dots v_n$.

The method due to Wagner and Fischer[12] is used to compute the minimum distance $d(u, v)$ between two strings $u = u_1 \dots u_m$ and $v = v_1 \dots v_n$, obtaining an $m \times n$ matrix M in which $M[i, j]$ gives the minimum distance between $u_1 \dots u_i$ and $v_1 \dots v_j$. The algorithm given by Wagner and Fischer actually computes least cost. It is simplified to compute minimum distance, by letting all edit costs take the value 1.

As an example grammar we use a small language for simple arithmetic expressions generated by the grammar

$$G = (\{E, T, F\}, \{id, +, *, (,)\}, P, E) \tag{1}$$

where P contains the productions

1. $E \rightarrow T$
2. $E \rightarrow E + T$
3. $T \rightarrow F$
4. $T \rightarrow T * F$
5. $F \rightarrow id$
6. $F \rightarrow (E)$

Consider the input string $id\ *\ (\ id\ id\ id\ )\ +\ id\ +\ id$. Taking τ to be 6 and σ to be 3, the next τ input symbols after detection of error at the fifth input symbol are $id\ id\ )\ +\ id\ +$. We compare just two of the possible continuation strings, $+\ id\ +$ and $+\ id\ )$. Table I shows the minimum distance matrices for the actual input string $id\ id\ )\ +\ id\ +$ and the two continuation strings $+\ id\ +$ and $+\ id\ )$.

Although the minimum distance between $id\ id\ )\ +\ id\ +$ and $+\ id\ +$ is 3, less than the distance of 4 between $id\ id\ )\ +\ id\ +$ and $+\ id\ )$, the latter string is a better repair because it has a distance of 1 from the prefix $id\ id\ )$ of the remaining input, indicated by the smallest entry in the last rows of the two matrices. The string $+\ id\ )$ is therefore chosen to replace the prefix $id\ id\ )$ of the remaining input, giving a repaired string $id\ *\ (\ id\ +\ id\ )\ +\ id\ +\ id$.

No practical method can guarantee to find a minimum-distance error correction, as there may be an arbitrary number of input symbols to inspect before the correct choice of repair can be made. On the other hand, increasing the amount of lookahead on the input, or the length of continuation strings to be generated, should improve the chance of making the correct choice of repair. Both these tactics supply more information to be used in making that choice. Increasing the number of input symbols to be inspected means that for some inputs, enough symbols will be seen to make the best choice. Increasing the length of continuation strings means that repairs which diverge from the actual input can be discarded.

Minimum Distance Recovery for LR Parsers

The problem of generating continuation strings is solved with the use of the LR parse tables. For any state of an LR parser, the tables indicate which input symbols give rise to a legal move, either shift, reduce or accept. The concept of a *recovery configuration* is used to model the successive concatenation of such legal symbols to a continuation string. A recovery configuration consists of an LR parser stack of states and a continuation string, analogous with a conventional configuration of a stack of states and unexpended input. An initial recovery configuration consists of the parser stack at the point of detection of error and the empty string. Successive recovery configurations are formed from each legal (shift, reduce or accept) move. A shift move gives rise to a recovery configuration consisting of the stack with the shift state pushed on and the shift symbol concatenated to the end of the continuation string. A reduce move gives rise to a configuration consisting of the reduced stack and the (previous) continuation string. An accept move on the input endmarker symbol $ does not give rise to further configurations, but indicates that the continuation string is a suffix of the consumed input, i.e. the consumed input concatenated with the continuation string forms a sentence of the language.

Some notation for LR parsers is required. Each state q in the set of states Q of an LR(1) parsing machine is constructed from a set of LR(1) items of the form $[A \rightarrow \alpha \cdot \beta, a]$ where $A \rightarrow \alpha\beta$ is a production of a context-free grammar $G = (N, \Sigma, P, S)$ and a in Σ is a terminal of the grammar. A configuration of an LR parser is represented by an instantaneous description (ID) $[q_0 \ldots q_m, a_j \ldots a_{j+k}]$, where $q_0 \ldots q_m$ is the sequence of states on the parsing stack with q_m at the top, and $a_j \ldots a_{j+k}$ is the unexpended input. Moves of an LR(1) parser are given by the ACTION and GOTO transition functions

ACTION: $Q \times \Sigma \cup \{\$\} \rightarrow \{\text{SHIFT}\} \times Q \cup \{\text{REDUCE}\} \times P \cup \{\text{ACCEPT}\} \cup \{\text{ERROR}\}$

GOTO: $Q \times N \rightarrow Q$

The relation *move in one step* on parser configurations, denoted by the symbol $\vdash$, is defined as follows.

If $\text{ACTION}(q_m, a_j) = (\text{SHIFT}, q)$, then $[q_0 \ldots q_m, a_j \ldots a_{j+k}] \vdash [q_0 \ldots q_m q, a_{j+1} \ldots a_{j+k}]$.

If $\text{ACTION}(q_m, a_j) = (\text{REDUCE}, A \rightarrow \alpha), |\alpha| = n$, and $\text{GOTO}(q_{m-n}, A) = q$, then $[q_0 \ldots q_m, a_j \ldots a_{j+k}] \vdash [q_0 \ldots q_{m-n} q, a_j \ldots a_{j+k}]$.

A recovery configuration of an LR(1) parser is represented by an ID $\langle q_0 \ldots q_m, u \rangle$, where $q_0 \ldots q_m$ is the sequence of states on the parsing stack with q_m at the top, and u in $\Sigma*$ represents the continuation string. (Angle brackets $\langle$ and $\rangle$ are used in place of [and] in order to distinguish IDs representing recovery configurations from IDs representing conventional configurations.) The relation *move in one step* on recovery configurations, denoted by the symbol $\vdash$, is defined analogously to $\vdash$ on conventional configurations, as follows.

If $\text{ACTION}(q_m, a) = (\text{SHIFT}, q), a \in \Sigma$, then $\langle q_0 \ldots q_m, u \rangle \vdash \langle q_0 \ldots q_m q, ua \rangle$.

If $\text{ACTION}(q_m, a) = (\text{REDUCE}, A \rightarrow \alpha), |\alpha| = n$, and $\text{GOTO}(q_{m-n}, A) = q$, then $\langle q_0 \ldots q_m, u \rangle \vdash \langle q_0 \ldots q_{m-n} q, u \rangle$.

Let $\vdash^*$ denote the reflexive and transitive closure of $\vdash$. Let the configuration of the parser at the point of detection of error be given by the ID $[q_0 \ldots q_m, a_j \ldots a_{j+k}]$. Then the set Θ_σ of continuation strings of length up to σ symbols is given by

$\Theta_\sigma = \{b_i \ldots b_{i+\sigma-1} \mid b_j \in \Sigma, \langle q_0 \ldots q, \epsilon \rangle \vdash^* \langle q_0 \ldots q_n, b_i \ldots b_{i+\sigma-1} \rangle\} \cup \{b_i \ldots b_{i+k} \mid b_j \in \Sigma, 0 \leq k < \sigma - 1, \langle q_0 \ldots q_m, \epsilon \rangle \vdash^* \langle q_0 \ldots q_n, b_i \ldots b_{i+k} \rangle \text{ and } \text{ACTION}(q_n, \$) = \text{ACCEPT}\}$.

The set Θ_σ of continuation strings of length up to σ symbols is generated and the continuation string x whose minimum distance from a prefix of a fixed number τ of the unexpended input symbols is minimum over the set is chosen as the repair and used to replace the input prefix. The repair x in Θ_σ satisfies the following.
Let $\delta = d(x, a_j \ldots a_{j+l}) = \min\{d(x, a_j \ldots a_{j+i}) \mid i = 0, \ldots, \tau - 1\}$.
Then $\delta \leq d(y, a_j \ldots a_{j+i})$ for all y in $\Theta_\sigma, i = 0, \ldots, \tau - 1$.
Recovery returns control to the parser in the configuration $[q_0 \ldots q_m, xa_{j+l+1} \ldots a_{j+k}]$.

Figure 1 gives the scheme developed as a procedure *Recover* in pseudo-Pascal for use by an LR parser. Generation of the set of continuation strings is by the recursive procedure *GenerateRepairs*. *RepairString* records the best continuation string so far, that is the string of required length σ whose minimum distance from a prefix of the up-coming input is smallest of all continuation strings generated so far. It is not necessary to store the entire set of continuation strings as the method is only interested in the one nearest to the actual input. *RepairDistance* records the best minimum distance so far and *RepairLength* records the length of the prefix of input to be replaced by the best continuation string.

Examples

As an example we use the string $id \ * \ (\ id \ id \ id \ ) \ + \ id \ + \ id$ used previously. The error message issued by the implementation of the Minimum Distance Recovery Method described below is

```
Line 1: syntax error
id * ( id id id ) + id + id
----------^    replace 'id' with '+'
```

We show how this recovery action is synthesized by displaying the continuation strings generated and their minimum distances from the remaining input.

An LALR(1) parse table with default reductions (replacing error entries with reductions) for this grammar is shown in Table II. Blank entries in the ACTION section of the table are ERROR entries. Blank entries in the GOTO section are never consulted (don't care entries).

The parser detects an error when it is in configuration $[02758, id \ id \ ) + \ id \ + id \ \$]$. Generation of repairs commences with recovery configuration $\langle 02758, \epsilon \rangle$. Table III shows recovery configurations for generation of continuation strings of length 3. The left-hand column contains a configuration number which indicates how the configuration is generated, e.g. configuration (1.2) gives rise to configurations (1.2.1) and (1.2.2).

Table IV shows each continuation string generated together with the last row of its minimum distance matrix from the remaining input string $id \ id \ ) \ + \ id \ +$. The smallest entry in this table is minimum distance 1 between the continuation string $+ \ id \ )$ and the input prefix $id \ id \ )$, obtained by replacing the parser-defined error symbol id by $+$. Control is returned to the parser with configuration $[02758, + \ id \ ) \ + \ id \ + \ id \ \$]$.

Termination and Complexity

Termination of the parser with error recovery is assured because the error recovery procedure leaves the parser in a configuration in which the upcoming input consists of a repair string that will be consumed followed by a proper suffix of the original remaining input. The recovery algorithm has complexity $O(n)$, because the recursive procedure *GenerateRepairs* contains a modified version of the LR parsing algorithm which can be invoked up to a constant number ($|\Sigma|$) of times for each activation of *GenerateRepairs*.

Error Messages

We now show how the automatic generation of error messages can be achieved. We wish to supply the user with information about symbols which are in error and symbols which are legal alternatives. The first kind of information is supplied by a message indicating the parser-defined error symbol, of the form

```
Line 32, syntax error detected:
while a[1 do begin
----------^
```

The second kind of information could be supplied by a list of all admissible symbols at the point of detection of error. Such a list might be very long and we consider it more efficient and informative to the user to tell him or her exactly which symbols are chosen by the recovery method as the legal alternatives. This information can be supplied by a message of the form

```
']' inserted before 'do'
```

or

```
missing ']'
```

or one of the form

```
Line 32 replaced by 'while a[1] do begin'
```

The first form of message describes the action taken by the recovery method. The second form gives the same information as the first, but is phrased in terms of what the user has done rather than the recovery taken. The third message describes the action taken but gives the whole line rather than single symbol information. The second form of message may be more helpful to a beginning user, and less "egotistic";[14,15] but if the recovery method has chosen the wrong repair, the message will be confusing to a beginner and annoying to a knowledgeable user.

Another approach to providing information about legal alternatives is that taken by Sippu and Soisalon-Soininen,[16] whose method generates messages of the form

```
A symbol which could follow an expression has been inserted.
```

A message giving information about which symbols are in error can be emitted before the recovery method is invoked. A message giving information about which correct symbols have been chosen can be synthesized by the recovery method from the actions taken while recovery is performed, taking one of the forms

```
insert ']'
delete 'do'
replace ']' with '+'
```

The input to a parser usually consists of values representing lexical tokens supplied by a lexical analyser which processes the source input. Thus a parser has to construct messages from internal values representing the symbols of the CFG. An earlier approach[17] was to use the names given in the input specifications to *yacc* corresponding to those values, giving messages of the kind

```
LEFTBRACKET inserted before DO
```

However, the example messages above are phrased chiefly in terms of source input, which is clearer for the user who may not be familiar with the names of the grammar specification. In order to achieve this it is necessary for the message-generator to have information about source representations for the vocabulary symbols. The approach we have used is to associate with each token produced by the lexical analyser a string of characters forming the source input consumed by the lexical analyser to produce that token. With this interface the parser typically receives from the lexical analyser a triple consisting of a token value, a semantic value, and a source string.

This mechanism means that in error recovery, a repair involving actual input symbols (tokens) can have an associated synthesized message which is expressed in terms of source characters. A repair may also involve insertion of tokens. A representation of an inserted token as a string of source characters can be constructed by a lexical-analyzer generator from the regular expression used to define that token in the specification.

As examples of error messages, we show two Pascal programs from the Ripley collection of student programs,[18] together with the diagnostic output from the Minimum Distance Recovery Method and from the Berkeley Pascal compiler, whose recovery is based on the work of Graham, Haley and Joy.[19] Only the diagnostics which relate to syntax errors are shown for the Berkeley compiler; diagnostics relating to errors not described by the context-free syntax, such as type errors, declarations out of place or undefined variables, are not shown.

The first program contains a simple error: a semicolon is missing from the end of line 2.

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

The diagnostic output from the parser with the Minimum Distance Recovery Method is:

```
Line 3: syntax error
               if number > 1
---------------^
insert ';'
```

The diagnostic output from the Berkeley Pascal compiler is:

```
3                  if number > 1
e ----------------^--- Inserted ';'
```

The second program contains a more complex error: the formal parameter list does not include the type of the parameter.

```
program p(input,output);
procedure factorial (a);
    var q:integer; begin x:=1 end; begin end.
```

The diagnostic output from the parser with the Minimum Distance Recovery Method is:

```
Line 2: syntax error
procedure factorial (a);
----------------------^ insert ':'
-----------------------^ insert 'identifier'
```

The diagnostic output from the Berkeley Pascal compiler is:

```
2    procedure factorial (a);
E -------------------------^--- Expected ':'
E -------------------------^--- Inserted identifier
```

The first program, containing a simple error, gives rise to almost identical messages generated by the two systems. The second program shows a very slight difference in approach; the Minimum Distance Recovery Method notes the action taken in recovery, and the Berkeley compiler additionally notes an expected symbol. Preference between the two is a matter of taste.

Evaluation of Recovery Methods

Criteria for Error Handling

Following Horning,[1] Rörich[20] and Spenke et al,[21] we set several criteria for error handling, in the general areas of quality of performance, efficiency and ease of use. A good scheme should: detect all errors in the input; parse all the input; repair simple errors and recover from complex errors; generate good error messages; have practical requirements in time and space; have no effect on the analysis of correct input; be capable of automatic generation; be capable of incorporation into a practical parser-generator.

A parser incorporating the Minimum Distance Recovery Method detects all errors in the input and parses all the input. The method repairs every simple error and recovers from a complex error in the sense that it terminates and returns control to the parser. Using the scheme described above to generate error messages ensures that all error messages tell the user exactly which symbol lies at the point of detection of error, and the recovery action taken. All messages are directed towards the user and expressed in terms which the user understands.[22] They meet the criteria of simplicity, honesty and reliability.[15,23] They can be criticized for detailing the action taken — expressing the compiler's point of view — rather than telling what the user has done. We have used this approach because the recovery method cannot guarantee either to make a correct inference about the user's intention or to make a correct repair.

The Minimum Distance Recovery Method is practical in both time ($O(n)$) and space, requiring only a minor amount of storage for string representations of the CFG terminal symbols, in addition to the usual storage for LR parsers. It has been used in language compilers which run with normal memory requirements and with acceptable response times on various computers (DEC VAX/750, SUN 3 and SUN 4). There is no effect on the analysis of correct input, as the recovery method is only invoked when the parser detects an error. The parser proceeds exactly as normal on correct input, with no overheads.

The error handling scheme can be completely automatically generated: no additional specifications are required of the user other than the CFG and semantic actions required by the parser generator.

The scheme has been incorporated into the practical parser-generator *yacc*. In practice, with this parser-generator, fewer specifications are required than formerly, as the user no longer has to supply error productions if error recovery is required.

Performance in Practice

It is essential to evaluate the performance in practice of an error recovery scheme which is intended for practical use, and to use a representative collection of inputs for that evaluation. The Minimum Distance Recovery Method was incorporated in the LR parser-generator *yacc* by Holloway[24] and Dain.[25] The implementation was parameterized for the length σ of continuation strings generated and the amount of lookahead τ on the input. The interface between the lexical analyzer and the parser was implemented for *yacc* and its companion scanner-generator *lex* by Holloway.

Although the parser-generator incorporating the Minimum Distance Recovery Method can be used to build parsers with error recovery for any language which can be described by a *yacc* specification (any context-free language), we have concentrated performance evaluation on the programming languages Pascal and C. There are two reasons for this choice. Firstly, the primary aim of our work is to improve error recovery in programming language compilers, and Pascal and C are programming languages in very widespread use. Secondly, many authors have used the Ripley database[18] of student Pascal programs to evaluate performance, so comparisons of our scheme with others will be possible for Pascal. In addition to Pascal and C, we have constructed parsers for various other languages including C++[26] and awk.[27]

Parsers for C and Pascal were constructed using the implementation of the parser-generator *yacc* incorporating the Minimum Distance Recovery Method with the following pairs of values for σ, the length of strings generated, and τ, the amount of lookahead on the input: 4 and 8, 5 and 10, 6 and 12, 7 and 14. The grammar used for C was the ANSI C draft grammar. The grammar used for Pascal was the grammar of the Berkeley Pascal compiler.

The Ripley database of student Pascal programs was used as inputs for the Pascal parsers. The database consists of a reduced sample of original input programs, shortened and altered (presumably to keep the size of the database small). Associated with some of the errors is a weight that indicates how many times the particular kind of error occurred in the original sample. 121 of the Ripley programs contain one or more syntax errors according to the grammar we used. (The remaining few programs contain errors such as declarations in incorrect order which are not described by the grammar.) A collection of all C programs submitted to the C compiler on the University of Warwick Department of Computer Science VAX/75 over three separate 24 hour periods during October 1985 was made. At that time in the University calendar, there were many undergraduate students starting to learn the C language, so there were many short, erroneous programs submitted. All programs containing syntax errors and less than 100 lines long were used as inputs, making a total of 112 such programs altogether.

A Formal Method for Performance Evaluation

The aim of our new method for performance evaluation is to provide an objective measure for evaluation that uses formal definitions of errors. The method measures, for each input program, the number of minimum-distance errors and the number of edit operations performed by the error recovery scheme. The number of minimum-distance errors in a program is given by the global minimum-distance error correction according to the model of Aho and Peterson,[5] determined by inspection of the input. The number of edit operations performed by the recovery method is determined by running the parser with error recovery on the input and inspecting the resulting error messages, which detail the edit operations made. A comparison can then be made between the number of minimum-distance errors, the ideal, with the number of edit operations, the actual performance. Each input program is associated with an entry in a matrix showing numbers of minimum-distance errors against numbers of edit operations. For a given recovery method incorporated in a parser for a language, a table is drawn up for a set of input programs in the language, showing the total numbers of programs corresponding to each position in the matrix.

Tables were drawn up for both collections of programs, the Ripley set of Pascal programs (weighted and unweighted) and our set of C programs, for the Minimum Distance Recovery Method with the various values for σ and τ. Results are not available for the set of C programs with σ of 6 and 7, as error recovery took too long to complete with these parameters.

In order to use the formal evaluation method to make comparisons with another recovery scheme, tables were also drawn up for a method implemented in the parser-generator *yacc* referred to here as the Two Stage Recovery Method.[17] This method consists of two stages, a local recovery followed if necessary by a phrase-level recovery. Local recovery consists of a single edit operation (replace, delete or insert) on the input at the point of error. This repair is tested by attempting a forward move of the parser on the repaired input. If the repair permits the parser to consume a further fixed number ρ of input symbols, the repair is deemed to be successful and control is returned to the parser, with the repaired input. If the forward move of the parser fails before ρ symbols are consumed, then the second stage takes place. In this stage, the parser is returned to the configuration in which it determined an error, and the set of LR items for the current state is used to guide a phrase-level recovery. The last item in the kernel of the set of items is chosen to provide a goal non-terminal. If its first component is given by $[A \rightarrow X_1 \ldots X_m \cdot X_{m+1} \ldots X_n]$, then m states are popped from the stack and the GOTO state for the new top of stack state and non-terminal A is pushed. Finally, the parser is put in a configuration in which either it can shift the next input symbol or there is no more input, by repeatedly either making a legal reduce move or deleting the next input symbol. A reduction by the production $A \rightarrow X_1 \ldots X_m X_{m+1} \ldots X_n$ has been simulated.

Results of performance evaluation using the formal method are given in Tables V to XII. Tables V to VIII show the results for the Minimum Distance Recovery Method with values for σ of 4, 5, 6 and 7 on the Ripley collection of Pascal programs. Tables IX and X show the results for the Minimum Distance Recovery Method with values for σ of 4 and 5 on the collection of C programs. Tables XI and XII show the results for the Two Stage Recovery Method with the value 5 for ρ on the Pascal and C collections. The columns of each table are indexed by the number of errors according to the minimum-distance measure, and the rows are indexed by the number of edit operations performed by the error recovery method. An entry shows how many programs were found in that category. Blank entries are zeroes. For example, the entry of 55 in column 1, row 1 of Table V shows that 55 programs out of the Pascal collection contain 1 minimum-distance error and require 1 edit operation by the Minimum Distance Recovery Method with $\sigma = 4$.

The ideal for an error recovery method is to detect the exact number of errors in the input program. This imprecise description is formalized by the number of minimum-distance errors equalling the number of edit operations performed by the recovery method. Thus the ideal is to have all (non-zero) entries in the tables of results occurring on the diagonal. Table XIII shows, for each collection of programs and each recovery method, the percentage of programs for which the number of minimum-distance errors equals the number of edit operations.

Although the ideal for an error recovery method is to detect the exact number of errors in the input program, it is also interesting to ask the question, for how many programs does recovery nearly achieve this ideal. Expressing this formally, for how many programs does the number of edit operations made by recovery equal the number of errors plus or minus one. (How many programs lie on the diagonal or one away in the tables of results?) The answer is given in Table XIV which shows, for each collection of programs and for each recovery method, the percentage of programs for which the number of minimum-distance errors equals the number of edit operations plus or minus one.

In addition to the figures on overall performance, it is useful to know how the recovery method performs on programs that contain only a single error, and programs that contain multiple errors. The Ripley collection contains 72 programs (60%) with a single error, according to the minimum-distance measure. If the weights are taken into account, 71% contain a single error.

The C collection contains 67 programs (60%) with a single error. Table XV shows the percentage of single-error programs for which the number of minimum-distance errors equals the number of edit operations, for each collection of programs and each recovery method. Table XVI shows the percentage of multiple-error programs for which the number of errors equals the number of edit operations.

One of the aims when designing the Minimum Distance Recovery Method was to improve on the overall performance of the Two Stage Recovery Method, by maintaining good performance on single errors and improving on phrase-level recovery when a single edit operation fails to produce an acceptable repair. Table XV shows that, for single-error programs in the Pascal collection, the Minimum Distance Recovery Method ($\sigma = 7$) achieves excellent recovery (number of edit operations equals number of minimum-distance errors) for 83% of the programs, compared with 90% for the Two Stage Recovery Method. Table XVI shows that, for multiple-error programs in the Pascal collection, the Minimum Distance Recovery Method achieves excellent recovery for 44% of the programs, compared with 21% for the Two Stage Recovery Method. So the Minimum Distance Recovery Method does indeed handle multiple errors better than the Two Stage Recovery Method, but does not handle single-error programs as well. Inspection of the Pascal single-error programs shows that, although there are several programs for which both recovery methods find the same single-operation repair, there are also two for which the Two Stage Recovery Method finds a single-operation repair but the Minimum Distance Recovery Method finds a repair of two edit operations. One of these programs is shown below.

```
1    program p(input, output);    begin if x = 1 then begin
2        writeln('end of sort')
3      else writeln('loop detected in input order');
4      x:=1 end.
```

The diagnostics from the Two Stage Recovery Method are:

```
Line 3: syntax error
    writeln('end of sort')
--------------------------^
'end' inserted.
```

The diagnostics from the Minimum Distance Recovery Method are:

```
Line 3: syntax error
  else writeln('loop detected in input order');
--^ replace 'else' with ';'
Line 4: syntax error
  x:=1 end.
----------^
insert 'end'
```

With generated repairs of length 4 and lookahead on the input of 8 symbols, the Minimum Distance Recovery Method computes two repairs

```
        ; writeln ( '                            end writeln ( '
```

which are both at distance 1 from the input

```
        else writeln ( ' loop detected in input order ' ) ;
```

It is a matter of luck that the Two Stage Recovery Method makes the (arbitrary) choice of the better repair and the Minimum Distance Recovery Method does not. Not until several symbols later in the actual input is it apparent that the repair made by the Minimum Distance Recovery Method leads to a further edit operation.

Performance for the Pascal weighted set is generally better than for the unweighted set because the programs with higher weights are generally those containing simple single-token errors, which occurred more frequently in the original sample. (Tables XV and XVI show that both recovery methods perform better on single-error programs than on multiple-error programs.) The Minimum Distance Recovery Method ($\sigma = 4$) achieves similar performance for the Pascal unweighted set and the C set: excellent recovery for 67% of Pascal programs and 66% of C programs. When the length σ of repair generated is increased from 4 to

7, performance is improved slightly: from excellent recovery on 76% of Pascal (weighted) programs, up to 80%.

To summarize these results, the best overall performance for Pascal is obtained by the Minimum Distance Recovery Method with generated repairs of length 7 ($\sigma = 7$), achieving the theoretical ideal for 67% of the unweighted Pascal set and 80% of the weighted Pascal set. The method gives slightly better performance on Pascal than on C. Increasing the length of generated repair improves performance.

Comparison with Other Methods

The general approach of the Minimum Distance Recovery Method may be likened to that of Rörich,[20] who describes methods for error recovery in LL and LR parsers. Rörich's method finds a valid continuation string for the parsed input, and deletes symbols from the remaining input until an anchor symbol is met, that is any input symbol which is contained in the continuation string. The appropriate prefix of the continuation string is inserted and parsing continues from the anchor symbol. A major difference is that the Minimum Distance Recovery Method generates a number of continuations and chooses the repair from these, whereas Rörich's method generates only one continuation.

Our method for performance evaluation is qualitatively different from methods used by other authors, in that it is a formal method using a mathematical measure of performance instead of criteria such as "exactly what a competent programmer might have done", "no undesirable side-effects"[16] or "most plausible repair".[28]

The advantages of the method used here lie in its formality: it is precise, objective, and could be automated (although we have not done so). A potential disadvantage also lies in the method's formality: if the model of minimum-distance errors were not to correspond closely with the human user's concept of syntax errors, the method would provide an inappropriate measure of performance. We claim that the models are valid. The justification for this claim is given by analysis of the Ripley suite of Pascal programs, which shows that the minimum-distance error correction corresponds with the human's perception (as given by Ripley and Druseikis[18]) in 106 of the 121 programs (88%).

The advantage of methods using informal criteria such as those cited above is that the model of syntax errors is a human one (the evaluator's own model). The disadvantage of such methods is that the criteria are imprecise, subjective, and open to different interpretations. In the case of single-error programs, it is often straightforward to decide on the repair a competent programmer might make, for example insertion of a missing semi-colon. But difficulties are frequent with multiple-error programs. For example, one of the Ripley Pascal programs contains the fragment

```
const limit = 100; limitp1 = limit + 1;
```

This might be corrected by a programmer to

```
const limit = 100; limitp1 = 101;
```

using knowledge of both syntax rules (constant declarations may not contain expressions) and simple arithmetic. (It is also possible that questions would be asked about the necessity for declaring a second constant.) Ripley and Druseikis describe this error as an "extra + 1" but a competent programmer would be unlikely to correct it simply by deleting those extra symbols. Again, Ripley and Druseikis describe the error in the statement

```
hs := sqrt(2*pi*x)*x**x*exp(-x)
```

as “no exponentation operator in Pascal”. The error might be corrected by the competent programmer by a deletion of the extra multiplication sign, but it is plausible that he or she would recognize the intended meaning and suggest replacement with a function call. It is very difficult to decide on “the most plausible repair”. Firstly, how much knowledge will the programmer use in making the repair? Secondly, the Ripley suite consists of shortened and altered programs, so that they are not in themselves plausible programs; hence it is not possible to find a plausible repair. Stirling[28] notes that there are indeed cases where this criterion cannot be used, in which case he uses instead the minimum distance criterion. We claim that the minimum distance criterion provides an appropriate and accurate measure for all cases.

In order to compare our results of performance evaluation with those of other methods, we define the recovery action taken on a program to be *excellent* if the number of edit operations e equals the number of minimum-distance errors m. We equate this with the classification *excellent* of Sippu and Soisalon-Soininen[16] and the classification *good* of Stirling.[28] We define recovery to be *acceptable* if $e = m \pm 1$ and equate this with the combined classifications *excellent* and *good* of Sippu and Soisalon-Soininen, and the combined classifications *good* and *marginal* of Stirling.

Several authors[9,16,17,21,28,30,31,32,33] have used their error recovery methods in constructing syntax analysers for Pascal which are then tested on the Ripley suite. Table XVII shows the percentage of recovery actions performed by these schemes on the Ripley suite which are acceptable, and Table XVIII shows the percentage which are excellent.

The majority of methods give acceptable performance for 70 to 80% of Pascal programs. (It should be noted that the Minimum Distance Recovery Method also gives acceptable performance in this range for C programs, e.g. 79% with $\sigma = 5$.) The Minimum Distance Recovery Method gives acceptable performance for 80% or more of Pascal programs, performance which is bettered by the schemes of Spenke et al[21] and Burke and Fisher.[29,30] The high performance of these schemes and that of Boullier and Jourdan[31] may be partly due to hand-tuning of the recovery schemes for Pascal, as they employ language-specific data on input symbols.

Conclusions

A method for error recovery has been presented, using a single stage for repair of the input which aims to provide a practical minimum-distance repair, i.e. locally minimum-distance in linear time. The method satisfies established criteria for error handling schemes. The LR parser-generator *yacc* has been enhanced with the method and used to build compilers for several languages. A formal method for performance evaluation has been presented and used to evaluate the recovery scheme and one other. The scheme compares reasonably well with previous methods in terms of performance, and is superior in terms of its practical application, as its use is totally automated and is independent of language. The generation of clear, informative error messages has been automated. The resulting tools should be of interest and practical use to compiler writers.

References

[1] J. J. Horning, 'What the compiler should tell the user', in F. L. Bauer and J. Eickel (ed.), *Compiler Construction, An Advanced Course*, Springer Verlag, New York, 1974.

[2] S. Sippu, 'Syntax error handling in compilers', Report A-1981-1, Department of Computer Science, University of Helsinki, Helsinki, 1981.

[3] K. Hammond and V. J. Rayward-Smith, 'A survey on syntactic error recovery and repair', *Computer Languages*, **9**, 51–67 (1984).

[4] J. A. Dain, 'Syntax error handling in language translation systems', Research Report 188, Department of Computer Science, University of Warwick, Coventry, 1991.

[5] A. V. Aho and T. G. Peterson, 'A minimum-distance error-correcting parser for context-free languages', *SIAM Journal of Computing*, **1**, 305–312 (1972).

[6] D. Gries. 'Error recovery and correction, an introduction to the literature', in F. L. Bauer and J. Eickel (ed.), *Compiler Construction, An Advanced Course*, Springer Verlag, New York, 1974.

[7] G. Lyon. 'Syntax-directed least-errors analysis for context-free languages: a practical approach', *Communications of the ACM*, **17**, 3–14 (1974).

[8] T. Krawczyk. 'Error correction by mutational grammars', *Information Processing Letters*, **11**, 9–15 (1980).

[9] S. O. Anderson and R. C. Backhouse, 'Locally least-cost error recovery in Earley's algorithm', *ACM Transactions on Programming Languages and Systems*, **3**, 318–347 (1981).

[10] J. Mauney and C. N. Fischer, 'An improvement to immediate error detection in strong LL(1) parsers', *Information Processing Letters*, **12**, 211–212 (1981).

[11] J. Earley, 'An efficient context-free parsing algorithm', *Communications of the ACM*, **13**, 94–102 (1970).

[12] R. A. Wagner and M. J. Fischer, 'The string-to-string correction problem', *Journal of the ACM*, **21**, 168–173 (1974).

[13] T. G. Peterson, 'Syntax error detection, correction and recovery in compilers', Ph. D. thesis, Stevens Institute of Technology, 1971.

[14] P. J. Brown, ' "My system gives excellent error messages" — or does it', *Software — Practice and Experience*, **12**, 91–94 (1982).

[15] P. J. Brown, 'Error messages: the neglected area of the man/machine interface?', *Communications of the ACM*, **26**, 246-249 (1983).

[16] S. Sippu and E. Soisalon-Soininen, 'A syntax-error handling technique and its experimental analysis', *ACM Transactions on Programming Languages and Systems*, **5**, 656–679 (1983).

[17] J. A. Dain, 'Error recovery for yacc parsers', *Proceedings of the EUUG Autumn 1985 Conference*, Copenhagen, 1985.

[18] D. C. Ripley and F. C. Druseikis, 'A statistical analysis of syntax errors', *Computer Languages*, **3**, 227–240 (1978).

[19] S. L. Graham, C. B. Haley and W. N. Joy, 'Practical LR error recovery', *ACM SIGPLAN Notices*, **14**(8), 168–175 (1979).

[20] J. Rörich, 'Methods for the automatic construction of error correcting parsers', *Acta Informatica*, **13**, 115–139 (1980).

[21] M. Spenke, H. Muhlenbein, M. Mevenkamp, F. Mattern and C. Beilken, 'A language independent error recovery method for LL(1) parsers', *Software — Practice and Experience*, **14**, 1095–1107 (1984).

[22] B. Dwyer, 'A user-friendly algorithm', *Communications of the ACM*, **24**, 556–561 (1981).

[23] E. Kantorowitz and H. Laor, 'Automatic generation of useful syntax error messages', *Software — Practice and Experience*, **16**, 627-640 (1986).

[24] N. W. Holloway, 'MINDER: Minimum distance error recovery for Yacc parsers', Department of Computer Science, University of Warwick, Coventry, 1988.

[25] J. A. Dain, 'Automatic error recovery for LR parsers in theory and practice', Ph.D. thesis, University of Warwick, Coventry, 1990.

[26] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.

[27] A. V. Aho, B. W. Kernighan and P. J. Weinberger, *The Awk Programming Language*, Addison-Wesley, Reading, MA, 1988.

[28] C. Stirling, 'Follow set error recovery', *Software — Practice and Experience*, **15**, 239–257 (1985).

[29] M. G. Burke and G. A. Fisher, 'A practical method for syntactic error diagnosis and recovery', *ACM SIGPLAN Notices*, **17**(6), 67–78 (1982).

[30] M. G. Burke and G. A. Fisher, 'A practical method for LR and LL syntactic error diagnosis and recovery', *ACM Transactions on Programming Languages and Systems*, **9**, 164–197 (1987).

[31] P. Boullier and M. Jourdan, 'A new error repair and recovery scheme for lexical and syntactic analysis', *Science of Computer Programming*, **9**, 271–286 (1987).

[32] T. M. Pennello and F. DeRemer, 'A forward move algorithm for LR error recovery', *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, 1978.

[33] A. Pai and R. B. 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).

Table I: Minimum distance matrices

	ϵ	id	id	$)$	$+$	id	$+$
ϵ	0	1	2	3	4	5	6
$+$	1	1	2	3	3	4	5
id	2	1	1	2	3	3	4
$+$	3	2	2	2	2	3	3

	ϵ	id	id	$)$	$+$	id	$+$
ϵ	0	1	2	3	4	5	6
$+$	1	1	2	3	3	4	5
id	2	1	1	2	3	3	4
$)$	3	2	2	1	2	3	4

Table II: LR parse table for grammar (1)

STATE	ACTION						GOTO		
	id	(	)	+	$*$	\$	E	T	F
0	S4	S5					1	2	3
1				S6		ACC			
2	R1	R1	R1	R1	S7	R1			
3	R3	R3	R3	R3	R3	R3			
4	R5	R5	R5	R5	R5	R5			
5	S4	S5					8	2	3
6	S4	S5						9	3
7	S4	S5							10
8			S11	S6					
9	R2	R2	R2	R2	S7	R2			
10	R4	R4	R4	R4	R4	R4			
11	R6	R6	R6	R6	R6	R6			

Table III: Generation of continuation strings

Configuration Number	Stack	Continuation String
(1)	$0\ T\ 2 * 7\ (\ 5\ E\ 8$	ϵ
(1.1)	$0\ T\ 2 * 7\ (\ 5\ E\ 8\ )\ 11$	$)$
(1.2)	$0\ T\ 2 * 7\ (\ 5\ E\ 8 + 6$	$+$
(1.1.1)	$0\ T\ 2 * 7\ F\ 10$	$)$
(1.2.1)	$0\ T\ 2 * 7\ (\ 5\ E\ 8 + 6\ id\ 4$	$+\ id$
(1.2.2)	$0\ T\ 2 * 7\ (\ 5\ E\ 8 + 6\ (\ 5$	$+\ ($
(1.1.1.1)	$0\ T\ 2$	$)$
(1.2.1.1)	$0\ T\ 2 * 7\ (\ 5\ E\ 8 + 6\ F\ 3$	$+\ id$
(1.2.2.1)	$0\ T\ 2 * 7\ (\ 5\ E\ 8 + 6\ (\ 5\ id\ 4$	$+\ (\ id$
(1.2.2.2)	$0\ T\ 2 * 7\ (\ 5\ E\ 8 + 6\ (\ 5\ (\ 5$	$+\ (\ ($
(1.1.1.1.1)	$0\ E\ 1$	$)$
(1.1.1.1.2)	$0\ T\ 2 * 7$	$)\ *$
(1.2.1.1.1)	$0\ T\ 2 * 7\ (\ 5\ E\ 8 + 6\ T\ 9$	$+\ id$
(1.1.1.1.1.1)	$0\ E\ 1 + 6$	$)\ +$
(1.1.1.1.1.2)	$0\ E\ 1$	$)\ \$$
(1.1.1.1.2.1)	$0\ T\ 2 * 7\ id\ 4$	$)\ *\ id$
(1.1.1.1.2.2)	$0\ T\ 2 * 7\ (\ 5$	$)\ *\ ($
(1.2.1.1.1.1)	$0\ T\ 2 * 7\ (\ 5\ E\ 8$	$+\ id$
(1.2.1.1.1.2)	$0\ T\ 2 * 7\ (\ 5\ E\ 8 + 6\ T\ 9 * 7$	$+\ id\ *$
(1.1.1.1.1.1.1)	$0\ E\ 1 + 6\ id\ 4$	$)\ +\ id$
(1.1.1.1.1.1.2)	$0\ E\ 1 + 6\ (\ 5$	$)\ +\ ($
(1.2.1.1.1.1.1)	$0\ T\ 2 * 7\ (\ 5\ E\ 8\ )\ 11$	$+\ id\ )$
(1.2.1.1.1.1.2)	$0\ T\ 2 * 7\ (\ 5\ E\ 8 + 6$	$+\ id\ +$

Table IV: Continuation strings and minimum distances

	ϵ	id	id	$)$	$+$	id	$+$
$+ \, (\, id$	3	2	2	3	4	4	5
$+ \, (\, ($	3	3	3	3	4	5	5
$) \, \$$	2	2	2	3	3	4	5
$) * id$	3	2	2	3	4	3	4
$) * ($	3	3	3	3	4	4	5
$+ \, id \, *$	3	2	2	2	3	4	4
$) + id$	3	2	2	3	3	2	3
$) + ($	3	3	3	3	3	3	4
$+ \, id \,)$	3	2	2	1	2	3	4
$+ \, id \, +$	3	2	2	2	2	3	3

Table V: Numbers of Pascal programs, edit operations against errors
Minimum Distance Recovery Method, $\sigma = 4$

Number of edit operations	Number of minimum-distance errors								
	1	2	3	4	5	6	7	8	9
1	55	1							
2	5	15	2						
3	5	3	6						
4	2	3	2	2					
5	1	1	1		1				
6	1		2		1	1	1		
7						1	1		
8	1	1							
9				1					
10									
11						1			
12	1								
>12	1								
Total	72	24	13	3	2	3	2	0	0

Table VI: Numbers of Pascal programs, edit operations against errors
Minimum Distance Recovery Method, $\sigma = 5$

Number of edit operations	Number of minimum-distance errors								
	1	2	3	4	5	6	7	8	9
1	55	1							
2	7	15	2				1		
3	3	5	6						
4	1	2		2					
5	2		2		1				
6			3		1		1		
7	1					2			
8									
9	1	1							
10	1								
11									
12									
>12	1								
Total	72	24	13	3	2	3	2	0	0

Table VII: Numbers of Pascal programs, edit operations against errors
Minimum Distance Recovery Method, $\sigma = 6$

Number of edit operations	Number of minimum-distance errors								
	1	2	3	4	5	6	7	8	9
1	57	1	1						
2	8	16	2		1				
3		3	6			1			
4	2	2	1	1					
5	1	1	3	1	1				
6							2		
7	1					2			
8				1					
9		1							
10									
11									
12	2								
>12	1								
Total	72	24	13	3	2	3	2	0	0

Table VIII: Numbers of Pascal programs, edit operations against errors
Minimum Distance Recovery Method, $\sigma = 7$

Number of edit operations	Number of minimum-distance errors: 1	2	3	4	5	6	7	8	9
1	59	2	1						
2	6	16	3		1	1			
3		4	4	1					
4	2	1	1						
5	2		2	1	1				
6			1			1	2		
7						1			
8	1	1		1					
9	1								
10									
11									
12	1								
>12	1								
Total	72	24	13	3	2	3	2	0	0

Table IX: Numbers of C programs, edit operations against errors
Minimum Distance Recovery Method, $\sigma = 4$

Number of edit operations	Number of minimum-distance errors								
	1	2	3	4	5	6	7	8	9
1	55	3	1		2				
2	3	6	2						
3	2	1	6			1			
4	2		1	3					
5		2			4				
6	3	2		1					
7	2	2							
8						1			
9								1	
10			1		1		1		1
11			1			1			
12									
>12									
Total	67	16	12	4	7	3	1	1	1

Table X: Numbers of C programs, edit operations against errors
Minimum Distance Recovery Method, $\sigma = 5$

Number of edit operations	Number of minimum-distance errors								
	1	2	3	4	5	6	7	8	9
1	57	3	1						
2	3	9	3		2				
3	3		6			1			
4	2								
5				3	1				
6	2	2			3				
7		2		1					
8			1		1				
9			1			2		1	
10							1		1
11									
12									
>12									
Total	67	16	12	4	7	3	1	1	1

Table XI: Numbers of Pascal programs, edit operations against errors
Two Stage Recovery Method, $\rho = 5$

Number of edit operations	Number of minimum-distance errors								
	1	2	3	4	5	6	7	8	9
1	57								
2		10							
3	1		1						
4	1	1							
5	1	1	2		1				
6		1							
7		1	2						
8	2	2	1						
9			1						
10									
11		1							
12		1							
>12	10	6	6	3	1	3	2		
Total	72	24	13	3	2	3	2	0	0

Table XII: Numbers of C programs, edit operations against errors
Two Stage Recovery Method, $\rho = 5$

Number of edit operations	Number of minimum-distance errors								
	1	2	3	4	5	6	7	8	9
1	63								
2	1	12							
3			3						
4		2	1	1					
5	1		4		1				
6			1				1		
7							1		
8									
9		1							
10				1					
11									
12	4								
>12	2	1	3	2	6	2		1	1
Total	67	16	12	4	7	3	1	1	1

Table XIII: Percentage of programs with number of minimum-distance errors equal to number of edit operations

Recovery Method	Pascal	Pascal (Weighted)	C
Two stage, $\rho = 5$	57%	71%	73%
Minimum Distance, $\sigma = 4$	67%	76%	66%
Minimum Distance, $\sigma = 5$	65%	77%	65%
Minimum Distance, $\sigma = 6$	67%	74%	-
Minimum Distance, $\sigma = 7$	67%	80%	-

Table XIV: Percentage of programs with number of minimum-distance errors equal to number of edit operations plus or minus one

Recovery Method	Pascal	Pascal (Weighted)	C
Two stage, $\rho = 5$	57%	71%	75%
Minimum Distance, $\sigma = 4$	80%	86%	76%
Minimum Distance, $\sigma = 5$	81%	87%	79%
Minimum Distance, $\sigma = 6$	83%	86%	-
Minimum Distance, $\sigma = 7$	84%	90%	-

Table XV: Percentage of single-error programs with number of minimum-distance errors equal to number of edit operations

Recovery Method	Pascal	Pascal (Weighted)	C
Two stage, $\rho = 5$	90%	91%	97%
Minimum Distance, $\sigma = 4$	77%	86%	82%
Minimum Distance, $\sigma = 5$	77%	86%	85%
Minimum Distance, $\sigma = 6$	80%	89%	-
Minimum Distance, $\sigma = 7$	83%	90%	-

Table XVI: Percentage of multiple-error programs with number of minimum-distance errors equal to number of edit operations

Recovery Method	Pascal	Pascal (Weighted)	C
Two stage, $\rho = 5$	21%	29%	40%
Minimum Distance, $\sigma = 4$	52%	53%	42%
Minimum Distance, $\sigma = 5$	48%	55%	36%
Minimum Distance, $\sigma = 6$	48%	40%	-
Minimum Distance, $\sigma = 7$	44%	56%	-

Table XVII: Percentage of recovery actions which are acceptable

Recovery Method	Acceptable Recovery
Two Stage[17]	57%
Stirling (a)[28]	66%
Stirling (b)[28]	66%
Sippu and Soisalon-Soininen[16]	67%
Pennello and DeRemer[32]	70%
Wirth[28]	72%
Boullier and Jourdan[31]	77%
IBM Pascal/VS[21]	77%
Pai and Kieburtz[33]	77%
Anderson and Backhouse[9]	79%
Minimum Distance, $\sigma = 4$	80%
Minimum Distance, $\sigma = 5$	81%
Minimum Distance, $\sigma = 6$	83%
Minimum Distance, $\sigma = 7$	84%
Spenke et al[21]	91%
Burke and Fisher[30]	98%

Table XVIII: Percentage of recovery actions which are excellent

Recovery Method	Excellent Recovery
Sippu and Soisalon-Soininen[16]	36%
Stirling (a)[28]	38%
Stirling (b)[28]	40%
Pennello and DeRemer[32]	42%
Wirth[28]	45%
Pai and Kieburtz[33]	52%
Two Stage[17]	57%
Anderson and Backhouse[9]	57%
Minimum Distance, $\sigma = 5$	65%
Minimum Distance, $\sigma = 4$	67%
Minimum Distance, $\sigma = 6$	67%
Minimum Distance, $\sigma = 7$	67%
Boullier and Jourdan[31]	75%
Spenke et al[21]	77%
Burke and Fisher[30]	78%

```
procedure Recover;
begin
    let the parser configuration be given by the ID [q0...qm, aj...aj+k];
    RepairString := ε;
    RepairDistance := 2 * τ;
    GenerateRepairs(q0...qm, ε);
    set the parser configuration to [q0...qm, RepairString aj+RepairLength...aj+k]
end;

procedure GenerateRepairs(Stack, Continuation);
begin
    let Stack be denoted by the states q0...qm;
    let Continuation be denoted by the symbols b1...bn;
    if n = σ or ACTION(qm, $) = ACCEPT then
    begin
        for i := 0 to τ-1 do
            if MinDist(Continuation, aj...aj+i) < RepairDistance then
            begin
                RepairString := Continuation;
                RepairDistance := MinDist(Continuation, aj...aj+i);
                RepairLength := i+1
            end
    end
    else for each symbol b in Σ do
        if ACTION(qm, b) = (SHIFT, q) then
            GenerateRepairs(q0...qmq, b1...bnb)
        else if ACTION(qm, b) = (REDUCE, A → α) then
        begin
            p := |α|;
            q := GOTO(qm-p, A);
            GenerateRepairs(q0...qm-pq, b1...bn)
        end
end;
```

Figure 1: The Minimum Distance Recovery Method for LR parsing