THE UNIVERSITY OF
WARWICK

University of Warwick institutional repository: http://go.warwick.ac.uk/wrap

**A Thesis Submitted for the Degree of PhD at the University of Warwick**

http://go.warwick.ac.uk/wrap/4162

This thesis is made available online and is protected by original copyright.

Please scroll down to view the document itself.

Please refer to the repository record for this item for information to help you to cite it. Our policy information is available from the repository home page.

# Automatic Error Recovery for LR Parsers

# in Theory and Practice

Julia Anne Dain M.A.

Submitted in respect of the requirements

for the degree of Doctor of Philosophy

to the University of Warwick

Department of Computer Science

University of Warwick

September 1989.

# Table of Contents


**Prologue** 10

**Chapter 1 Introduction** 11

1.1 Introduction to the Problem 11

1.2 Error Handling, Recovery, Repair and Correction 13

1.3 Notation and Definitions 14

1.4 Models for Syntax Errors 17

1.4.1 Minimum-Distance Errors 18

1.4.2 Prefix-Defined Errors 21

1.5 Error Detection 25

**Chapter 2 Literature Survey** 27

2.1 Surveys and Bibliographies on Error Handling 27

2.2 Parsing, Parser Generators and Minimum Distance 28

2.2.1 LR Parsing 28

2.2.2 General Parsing 29

2.2.3 Parser Generators 29

2.2.4 String Comparison and Minimum Distance 30

2.3 Language Design 30

2.4 The User Interface - Error Reporting 33

2.5 Error Handling Schemes 34

2.5.1 Early Work 34

2.5.2 Schemes that Perform Global Correction 36

2.5.3 Schemes that Alter the Parsing Stack and the Input 37

2.5.4 Schemes that Alter the Input Alone 42

2.6 Evaluation 46

2.6.1 Performance Evaluation and Comparison 46

2.6.2 Automatic Generation 49
2.6.3 Questions Arising 50

**Chapter 3 Good Error Recovery 51**
3.1 Criteria for Error Recovery 51
3.1.1 Performance 52
3.1.2 Efficiency 54
3.1.3 Ease of Use 55
3.1.4 Goals 55
3.2 CFLs and Error Recovery 56
3.2.1 Minimum Distance between Sentences 56
3.2.2 Legal Continuations 58
3.2.3 Distance between Minimum-Distance and Prefix-Defined Errors 63
3.3 CFGs and Error Recovery 68
3.3.1 CFGs which Generate the Required Language 68
3.3.1 CFGs which Generate a Superset of the Required Language 69
3.4 Parsing Method and Error Recovery 70
3.4.1 Detection of Errors 70
3.4.2 Information Available at Point of Detection 72

**Chapter 4 Two Methods for Error Recovery 76**
4.1 Notation 76
4.2 Recovery Method 1 77
4.2.1 Outline 77
4.2.2 Recovery Method 1 for LR Parsing 79
4.2.3 Examples 84
4.2.4 Termination and Complexity 91
4.3 Recovery Method 2 92
4.3.1 Outline 92

4.3.2 Recovery Method 2 for LR Parsing 97
4.3.3 Examples 100
4.3.4 Termination and Complexity 103
4.4 The LR Parser-Generator *Yacc* 103
4.5 Comparison with Other Methods 106
4.6 Automatic Generation of Error Messages 107

**Chapter 5 Evaluation of Recovery Methods 112**
5.1 Satisfaction of Criteria for Error Handling 112
5.2 Performance Evaluation 115
5.2.1 Implementation 116
5.2.2 Parsers and Inputs 116
5.2.3 Method 117
5.2.4 Results 120
5.2.5 Analysis 121
5.3 Discussion 123
5.4 Comparison with Other Methods 131
5.4.1 Methods for Performance Evaluation 131
5.4.2 Results of Performance Evaluation 132

**Chapter 6 Conclusions 135**
6.1 Summary 135
6.2 Conclusions 135
6.3 Suggestions for Further Work 136
6.3.1 Alterations and Extensions 136
6.3.2 Future Directions 137

**Appendix A Extract from "Minimum Distance Error Correction" 141**

**Appendix B Parser Template** 149

**Appendix C Results of Performance Evaluation** 157

**Bibliography** 166

# List of Figures


| | | |
|---|---|---|
| **Fig. 1.1.** | An incorrect Pascal program | 22 |
| **Fig. 2.1.** | Percentage of recovery actions which are acceptable | 47 |
| **Fig. 3.1.** | SLR(1) parse table for grammar (3.1) | 73 |
| **Fig. 3.2.** | LL(1) parse table for grammar (3.2) | 74 |
| **Fig. 3.3.** | Parse trees for input *( a a* ... | 75 |
| **Fig. 4.1.** | Recovery Method 1 for LR parsing | 81 |
| **Fig. 4.2.** | LR parse table for grammar (4.1) | 85 |
| **Fig. 4.3.** | Repairs resulting from different grammars | 91 |
| **Fig. 4.4.** | Procedure to compute minimum distance | 95 |
| **Fig. 4.5.** | Minimum distance matrices | 96 |
| **Fig. 4.6.** | Recovery Method 2 for LR parsing | 99 |
| **Fig. 4.7.** | Generation of continuation strings | 101 |
| **Fig. 4.8.** | Continuation strings and minimum distances | 102 |
| **Fig. 4.9.** | Recovery method used by *yacc* | 104 |
| **Fig. 4.10.** | Pascal program containing a simple error | 110 |
| **Fig. 4.11.** | Pascal program containing a complex error | 111 |
| **Fig. 5.1.** | Error messages for a Pascal program containing one error | 113 |
| **Fig. 5.2.** | Error messages for a Pascal program containing two errors | 114 |
| **Fig. 5.3.** | Pascal program requiring several edit operations | 119 |
| **Fig. 5.4.** | Numbers of programs with same number of prefix-defined and minimum-distance errors | 120 |
| **Fig. 5.5.** | Numbers of programs containing a single error | 120 |
| **Fig. 5.6.** | Percentage of programs with number of minimum-distance errors equal to number of edit operations | 122 |
| **Fig. 5.7.** | Percentage of programs with number of minimum-distance errors equal to number of edit operations plus or minus one | 122 |

**Fig. 5.8.** Percentage of single-error programs with number of minimum-distance errors equal to number of edit operations 123

**Fig. 5.9.** Percentage of multiple-error programs with number of minimum-distance errors equal to number of edit operations 124

**Fig. 5.10.** Different repairs for a single-error program 125

**Fig. 5.11.** Different repairs for a single-error program 126

**Fig. 5.12.** Cluster of errors in a Pascal program 129

**Fig. 5.13.** Pascal program with largest weight 130

**Fig. 5.14.** Percentage of recovery actions which are acceptable 133

**Fig. 5.15.** Percentage of recovery actions which are excellent 134

**Fig. 6.1.** An interactive session with Standard ML 139

## Acknowledgements

I would like to record my most grateful thanks to Meurig Beynon, for supervising the writing of this thesis; to Mathai Joseph and Mike Paterson, for several useful discussions; to Nick Holloway, for his implementations; to Mark Rafter and Jeff Smith, for their practical help; and to Derek Roberts, Anna Bayman and Rosie Bayman, for their encouragement and support.

## Declaration

The material contained in this thesis has not been used before, with the exception of Appendix A which is extracted from (Dain, 1987). The work presented is the author's own work, except for the implementations of Recovery Method 2 and of the interface between *lex* and *yacc*, which were carried out by Holloway.

# Summary

This thesis argues the need for good syntax error handling schemes in language translation systems such as compilers, and for the automatic incorporation of such schemes into parser-generators. Syntax errors are studied in a theoretical framework and practical methods for handling syntax errors are presented.

The theoretical framework consists of a model for syntax errors based on the concept of a minimum prefix-defined error correction, a sentence obtainable from an erroneous string by performing edit operations at prefix-defined (parser defined) errors. It is shown that for an arbitrary context-free language, it is undecidable whether a better than arbitrary choice of edit operations can be made at a prefix-defined error. For common programming languages, it is shown that minimum-distance errors and prefix-defined errors do not necessarily coincide, and that there exists an infinite number of programs that differ in a single symbol only; sets of equivalent insertions are exhibited.

Two methods for syntax error recovery are presented. The methods are language independent and suitable for automatic generation. The first method consists of two stages, local repair followed if necessary by phrase-level repair. The second method consists of a single stage in which a locally minimum-distance repair is computed. Both methods are developed for use in the practical LR parser-generator *yacc*, requiring no additional specifications from the user. A scheme for the automatic generation of diagnostic messages in terms of the source input is presented. Performance of the methods in practice is evaluated using a formal method based on minimum-distance and prefix-defined error correction. The methods compare favourably with existing methods for error recovery.

## Prologue

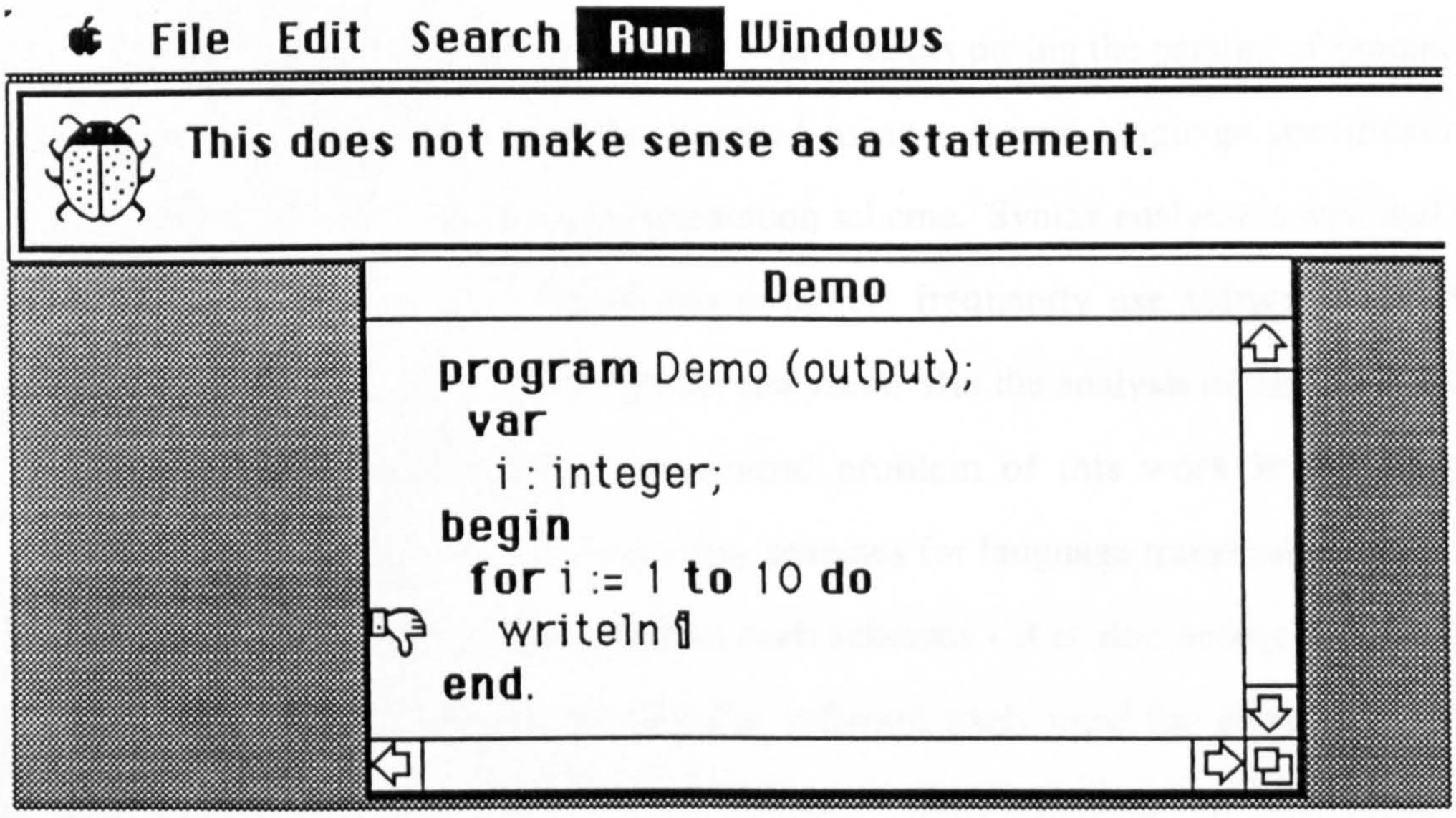
A programmer fluent in BCPL and Algol is working on a program in a new language, the C programming language. She submits the following program to the C compiler running under the UNIX system.

```
1       #include <stdio.h>
2       main()
3       {   int i;
4           i = 1
5           printf("%d\n", i)
6       }
```

The diagnostics produced by the compiler are:

```
"ex.c", line 5: syntax error
"ex.c", line 5: illegal character: 134 (octal)
"ex.c", line 5: cannot recover from earlier errors: goodbye!
```

A student with some BASIC programming experience is developing a Pascal program with MacPascal on an Apple Macintosh. He types in some text and tries to run the program, with the following result:




These stories, although imaginary, are drawn from actual experience; the facts in them are true and the errors made are common. Both stories show how unhelpful the diagnostics issued by current language translators can be. The work presented in this thesis originates in the author's experience as a programmer: use of a number of different compilers and interpreters gave rise to a conviction that error handling could be improved and to a desire to work in the area.

# Chapter 1 Introduction

## 1.1 INTRODUCTION TO THE PROBLEM

Use of a computer involves a dialogue between user and machine: the dialogue is usually formal, and is susceptible to human errors in the use of formal language. So the problem of handling errors in a dialogue faces every designer of a computer system. Good error handling enhances the value of a system by showing its users how to conduct a correct dialogue; bad error handling prevents effective use of a system, and in many cases, such as in safety-critical systems, may incur danger. The difficulties experienced by someone trying to conduct a dialogue without help when errors occur should not be part of computer usage.

Errors occur at several different levels of use of language, and the handling of errors is a large problem involving linguistic and psychological theories. This thesis is concerned with how to handle errors of syntax, a problem which occurs during the parsing of sentences in a dialogue. Dialogues are frequently designed using a formal language specification and implemented using a syntax-directed translation scheme. Syntax analysis is well understood for correct sentences, to the extent that designers frequently use software tools for the automatic construction of efficient syntax analysers. But the analysis of incorrect sentences remains less well understood. The central problem of this work is the design and construction of good syntax error recovery schemes for language translation systems. It is not sufficient just to be able to construct such schemes - it is also necessary to be able to construct them automatically, so that the software tools used for generating language translators can incorporate the schemes. If this problem can be solved, then dialogues that are ineffective, dangerous or frustrating because of inadequate syntax error recovery should indeed be a thing of the past.

The issues in syntax error handling, stated briefly, are: what is an error? how will it be dealt with? which languages will be handled? and what are the technical issues for the parser?

The first question, what is an error, can be answered formally, by giving a theoretical model which characterizes errors abstractly, or informally, from a practical or user-oriented view which characterizes errors according to plausibility or frequency of occurrence. The second question, how will an error be dealt with, can be answered either by halting syntax analysis as soon as an error is detected, or by attempting to continue analysis by some means of recovery. Recovery must restart the parser and may include alteration or repair to the input. It may not be feasible to produce a theoretically optimal repair, because of considerations of efficiency, nor may it be feasible to determine the original intention of the user, because of the nature of the system in which the recovery method operates. The third question is whether the languages to be handled are arbitrary context-free languages or a restricted class, such as practical programming languages. Typical programming languages may exhibit special characteristics which have implications for error handling. Studying such languages, and the use of them, may lead in different directions from a theoretical study: for example, to questions such as what are the psychologically plausible errors or the most common in practice, which errors are hard to handle, and what are the implications for programming language definition. The fourth question addresses the technical issues for a parser: how will errors be identified, how will recovery be effected, what communication with the user will take place. Relevant factors include the method used for parsing, the nature of access to the source input, the extent to which strategies are language-dependent, and the need for efficiency.

This thesis is a broad study of theory and practice of syntax error handling which addresses some of the above issues; in particular, the issues of theoretical models for syntax errors, theoretical properties of arbitrary context-free languages, and practical, language-independent methods for error recovery. The thesis is organized as follows. Chapter 1 introduces syntax error handling and presents a new theoretical model for syntax errors based on the concept of minimum prefix-defined error correction. Chapter 2 contains a survey of the literature on syntax errors, including categorization and evaluation of current techniques for recovery. Chapter 3 establishes criteria for judging error recovery and

presents theoretical results concerning languages and parser design which place limitations on the performance of an error handling scheme. Chapter 4 presents two new algorithms for error recovery and a scheme for generating diagnostic messages. These are implemented in a parser-generator which is used to build translation systems for several languages. Chapter 5 presents a new empirical method for performance evaluation and the results of evaluation of the two new error recovery schemes. Chapter 6 contains conclusions and suggestions for further work.

The focus for the work is syntax error handling in production translators. Most of the error messages generated by a language translation system relate to syntax errors in the source input. Treatments of lexical errors (spelling correction), errors in the context-sensitive grammar structure, semantic errors and logical errors lie outside the scope of this work. Because our primary motivation was to improve the quality of syntax error handling in current systems, which are typically batch-mode or non-interactive compilers and interpreters, we wished to build practical tools which could be used to generate such systems in a production environment. The context for the work is therefore that of existing compiler-writers' tools, particularly LR parsers and LR parser-generators.

Familiarity is assumed with the theory of formal languages and automata, as presented by Hopcroft and Ullman (1979), and with the principles and practice of compiler design, as presented by Aho and Ullman (1977).

## 1.2 ERROR HANDLING, RECOVERY, REPAIR AND CORRECTION

The terms error handling, error recovery, error repair and error correction are used with different meanings in the literature. In this work the following definitions will be used. *Error handling*, or an *error handling scheme*, is a general term for the method used to treat a syntax error in the input to a parser, including reporting the error to the user. *Error recovery* is error handling in which an attempt is made to continue parsing the remaining input after

detection of a syntax error. *Error repair* describes those changes which an error handling scheme may make to the input and possibly the parse stack in order to continue parsing. *Error correction* is error handling which alters the input to that which the user intended.

An error handling scheme may range from one which quits as soon as an error is detected, to one which performs minimum-distance error correction. Most error handling schemes attempt some form of error recovery, rather than quitting, looping or crashing. True error correction cannot be achieved for all inputs by any scheme which does not consult the user about his or her intentions, but many authors use the term *correction* with a less strict meaning. Conway and Wilcox (1973) use the term for a compiler that always produces translated code from a syntactically correct structure. Minimum-distance error correction is the term used to describe schemes in which the user's intention is modelled by the minimum distance metric (Aho and Peterson, 1972; Lyon, 1974; Krawczyk, 1980). The error handling in these schemes is integrated with parsing, rather than invoked as a separate module when an error is detected. Some schemes perform minimum-distance error correction over a bounded region (Anderson and Backhouse, 1981; Mauney and Fischer, 1982). Most schemes however are not claimed to perform error correction, but use some form of error repair to effect error recovery. The next chapter contains a detailed survey of error handling schemes.

## 1.3 NOTATION AND DEFINITIONS

This section presents notations, conventions and preliminary definitions of formal language theory, following Hopcroft and Ullman, and may be omitted by a reader with the necessary background.

The empty string is denoted by the symbol $\varepsilon$.

The empty set is denoted by the symbol $\emptyset$.

A context-free grammar (CFG) $G$ is a tuple $(N, \Sigma, P, S)$ where

$N$ and $\Sigma$ are disjoint finite sets of non-terminal and terminal symbols respectively,

$S$ in $N$ is a distinguished symbol called the start symbol,

$P$ is a finite set of productions of the form $A \rightarrow \alpha$ where $A$ is in $N$ and $\alpha$ is in $(N \cup \Sigma)^*$.

The vocabulary or set of grammar symbols $V$ of $G$ is $N \cup \Sigma$.

A pushdown automaton (PDA) $M$ is a system $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where

$Q$ is a finite set of states,

$\Sigma$ is the input alphabet,

$\Gamma$ is the stack alphabet,

$\delta$ is a mapping from $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$,

$q_0$ in $Q$ is the initial state,

$Z_0$ in $\Gamma$ is the stack start symbol,

$F$, a subset of $Q$, is the set of final states.

A configuration of a PDA is described by an instantaneous description $(q, w, \gamma)$ where

$q$ is a state,

$w$ is a string of unexpended input symbols,

$\gamma$ is a string of stack symbols.

The strings $w$ and $\gamma$ are written so that the next input symbol and the top stack symbol are at the leftmost end of the strings. If $\delta(q, a, Z)$ contains $(p, \beta)$, where $a$ may be an input symbol or $\varepsilon$, then $(q, aw, Z\alpha) \vdash (p, w, \beta\alpha)$.

A parser for a CFL $L$ is a PDA $M$ accepting $L$, where the input alphabet of $M$ is the set of terminals of $G$.

Unless otherwise stated, the following conventions will be adopted.

For a finite alphabet $\Sigma$,

lower-case $a, b, \ldots$ denote single symbols of $\Sigma$,

lower-case $u, v, \ldots$ denote strings over $\Sigma^*$.

For a CFG $G$,

lower-case $a, b, \ldots$ denote single terminals,

lower-case $u, v, \ldots$ denote strings of terminals,

upper-case $A, B, \ldots$ denote single non-terminals,

lower-case Greek $\alpha, \beta, \ldots$ denote strings of vocabulary symbols.

For a PDA $M$,

lower-case $a, b, \ldots$ denote single input symbols,

lower-case $u, v, \ldots$ denote strings of input symbols,

upper-case $Z, \ldots$ denote single stack symbols,

lower-case Greek $\alpha, \beta, \ldots$ denote strings of stack symbols.

It will be assumed that every CFL is non-empty, and, without loss of generality, that a CFG contains no useless symbols (every non-empty CFL is generated by a CFG with no useless symbols, Hopcroft and Ullman (1979), p. 89).

The concepts of prefix, suffix and quotient will be needed to describe various strings, for example input accepted by a parser and strings acceptable by a particular state of a parser.

*Defn 1.1* (Greibach, 1968) The *set of prefixes* Pre($L$) of a language $L$ is given by

$$\mathrm{Pre}(L) = \{\ u \mid uv \text{ is in } L \text{ for some } v \text{ in } \Sigma^*\ \}.$$

*Defn 1.2* (Greibach, 1968) The *set of suffixes* Suf($L$) of a language $L$ is given by

$$\mathrm{Suf}(L) = \{\ v \mid uv \text{ is in } L \text{ for some } u \text{ in } \Sigma^*\ \}.$$

*Defn 1.3* (Greibach, 1968) The *right quotient* $L_1/L_2$ of a language $L_1$ by a language $L_2$ is given by

$$L_1/L_2 = \{\ u \mid uv \text{ is in } L_1 \text{ for some } v \text{ in } L_2\ \}.$$

*Defn 1.4* (Greibach, 1968) The *left quotient* $L_2 \backslash L_1$ of a language $L_1$ by a language $L_2$ is given by

$$L_2 \backslash L_1 = \{\ u \mid vu \text{ is in } L_1 \text{ for some } v \text{ in } L_2\ \}.$$

If $L$ is a CFL then Pre($L$) and Suf($L$) are CFLs. If $R$ is a regular set then $L/R$ and $L \backslash R$ are CFLs (Greibach, 1968).

## 1.4 MODELS FOR SYNTAX ERRORS

Before discussing how to handle syntax errors, it is necessary to establish with some precision what a syntax error is; an informal concept of a syntax error does not necessarily correspond with a formal definition. A CFG used to describe the context-free syntax of a programming language defines precisely those strings which are syntactically correct programs, namely sentences of the language, but not what the syntax errors are for a string which is not a sentence of the language. This section reviews the model of minimum distance and presents a new model, prefix-defined error correction.

### 1.4.1 Minimum-Distance Errors

Minimum or Hamming distance is used by many authors (see for example Aho and Peterson (1972), Sippu (1981) and Gries (1975)) for a formal model which approximates the programmer's concept of syntax errors, as it measures the shortest way to transform a syntactically incorrect program into a correct one. Given a fixed set of edit operations, typically the insertion, deletion or replacement of a single symbol, the minimum distance between two strings is the minimum number of edit operations needed to transform one string into the other. Additionally, costs may be associated with the edit operations to give a least-cost sequence of edit operations needed to transform one string into the other (Wagner and Fischer, 1974). The minimum distance measure is used to define a minimum-distance error correction for a string and a language: a minimum-distance error correction is a sentence of the language nearest to the string, in the sense that there is no other sentence whose minimum distance from the string is smaller. A least-cost error correction is similarly defined to be a sentence of the language such that there is no other sentence whose least-cost sequence of edit operations is smaller. For syntax error correction, the symbols to be operated upon are the terminals of a CFG (the lexical tokens of a programming language).

Formally, let $\Sigma$ be a finite alphabet of symbols and let $\Delta$ be the set of transformations

$\{ (a, b) \mid a, b \in \Sigma \cup \{ \varepsilon \}, (a, b) \neq (\varepsilon, \varepsilon) \}$.

*Defn 1.5* (Wagner and Fischer, 1974) For strings $u, v$ in $\Sigma^*$, $u \rightarrow v$ *via* $a \rightarrow b$ in $\Delta$ if $(a, b) \in \Delta$ and there are strings $w, x$ in $\Sigma^*$ such that $u = wax$ and $v = wbx$.

*Defn 1.6* (Wagner and Fischer, 1974) $u \rightarrow v$ *via* $T$ if $T$ is a sequence of transformations $t_1 t_2 \ldots t_n$, $t_i = (a_i, b_i) \in \Delta$, and there are strings $w_1, w_2, \ldots, w_{n-1}$ in $\Sigma^*$ such that

$u \rightarrow w_1$ *via* $t_1$,

$w_i \rightarrow w_{i+1}$ *via* $t_{i+1}$ for $i = 1,\ldots, n\text{-}2$,

$w_{n-1} \rightarrow v$ *via* $t_n$.

*Defn 1.7* (Wagner and Fischer, 1974) The *minimum distance* $d(u, v)$ for strings $u, v$ is given by

$$d(u, v) = \min \{ n \mid u \rightarrow v \text{ via } T \text{ for some sequence } T = t_1 t_2 \ldots t_n \}.$$

*Defn 1.8* (Wagner and Fischer, 1974) For a cost function $\gamma : \Delta \rightarrow N$ the *cost* $\gamma(T)$ of a sequence of transformations $T = t_1 t_2 \ldots t_n$, $t_i \in \Delta$, is given by

$$\gamma(T) = \sum_{i=1}^{n} \gamma(t_i) .$$

*Defn 1.9* (Wagner and Fischer, 1974) The *least cost* $\delta(u, v)$ for strings $u, v$ is given by

$$\delta(u, v) = \min \{ \gamma(T) \mid u \rightarrow v \text{ via } T \text{ for some sequence } T \}.$$

*Defn 1.10* (Wagner and Fischer, 1974) A *minimum-distance error correction* for a string $u$ and a non-empty CFL $L$ over $\Sigma$ is a sentence $v$ of $L$ such that $d(u, v) \leq d(u, w)$ for all $w$ in $L$.

*Defn 1.11* (Wagner and Fischer, 1974) A *least-cost error correction* for a string $u$ and a non-empty CFL $L$ over $\Sigma$ is a sentence $v$ of $L$ such that $\delta(u, v) \leq \delta(u, w)$ for all $w$ in $L$.

The minimum-distance error correction and the least-cost error correction are not uniquely defined. As example we give the language $L(G)$, where

$G = ( \{ S \}, \{ (, ) \}, \{ S \rightarrow (), S \rightarrow (S) \}, S )$,

and the string *(()* . Two minimum-distance error corrections of distance 1 for this string are *()* and *(())*. This example is readily extended to that of common programming languages. For example, in Pascal (Cooper, 1983), arithmetic expressions may be bracketed with an arbitrary number of ( ) pairs, and in C (Kernighan and Ritchie, 1978), blocks may be bracketed with an arbitrary number of { } pairs.

A minimum-distance error correction for a string locates the errors at the points at which the edit operations are applied.

*Defn 1.12* The *minimum-distance error locations* for a string $u$ and a minimum-distance error correction $v$ for a CFL $L$ are after the strings

$x_1$ in $x_1 a_1 y_1$,

$x_{i+1}$ in $x_{i+1} a_{i+1} y_{i+1}$ for $i = 1,\ldots, n-1$,

$x_n$ in $x_n a_n y_n$,

where $u \rightarrow v$ *via* $T$ for the sequence $T = t_1 t_2 \ldots t_n$, $t_i = (a_i , b_i) \in \Delta$, and

$u = x_1 a_1 y_1$, $w_1 = x_1 b_1 y_1$,

$w_i = x_{i+1} a_{i+1} y_{i+1}$, $w_{i+1} = x_{i+1} b_{i+1} y_{i+1}$ for $i = 1,\ldots, n-2$,

$w_{n-1} = x_n a_n y_n$, $v = x_n b_n y_n$.

The location of the leftmost error in an incorrect string can be uniquely defined for a given minimum-distance error correction as the leftmost point at which the string differs from its correction.

*Defn 1.13* The *leftmost minimum-distance error location* for a string $u$ and a minimum-distance error correction $v$ for a CFL $L$ is after a prefix $w$ in Pre($L$) where

$$u = wx, \qquad v = wy,$$

and there is no $w'$ in Pre($L$) with $|w'| > |w|$ such that

$$u = w'x', \quad v = w'y'.$$

For a string with more than one correction we may wish to choose a correction with the longest possible prefix of the original string.

*Defn 1.14* A *longest prefix minimum-distance error correction* for a string $u$ and a CFL $L$ is a minimum-distance error correction $v$ where

$$u = wx, \qquad v = wy,$$

and there is no minimum-distance error correction $v'$ with

$$u = w'x', \quad v' = w'y', \quad \text{and } |w'| > |w|.$$

The leftmost error location for a longest prefix minimum-distance error correction then uniquely defines a point in an incorrect string which we call the *first minimum-distance error.* For a string and a longest prefix minimum-distance error correction at minimum distance one, there is only one point at which the string differs from the correction and this will be called the *minimum-distance error.*

### 1.4.2 Prefix-Defined Errors

A parser for a language may not detect an error in its input until some distance after the location of the leftmost error according to the minimum-distance error model. An example is given in Fig. 1.1, which shows an incorrect Pascal program in which the plausible error is the omission of the keyword '`for`' at the beginning of line 3. The minimum-distance error coincides with the plausible error and occurs on line 3 after the symbol '`begin`' at the

symbol 'n'.

```
1       program example(output);
2       begin
3          n := 1 to 10 do sum := sum + n
4       end.
```

**Fig. 1.1.** An incorrect Pascal program.

A minimum-distance error correction for the program is

```
1       program example(output);
2       begin
3          for n := 1 to 10 do sum := sum + n
4       end.
```

An LL or LR parser with one symbol of lookahead will not detect an error until the symbol 'to' is met, because the fragment 'n := 1' forms a legal Pascal statement. Peterson (1972) introduced the concept of parser defined errors to model this class of errors. In fact the errors are defined by the CFL and not by a parser for it and we shall therefore call these prefix-defined errors.

*Defn 1.15* (Peterson, 1972) The *prefix-defined error* in a string $u$ not in $L$ is after the string $x$ at the symbol $a$ where $u = xay$, $x$ is in Pre($L$) and $xa$ is not in Pre($L$).

We introduce the new concept of a *prefix-defined error correction* to model corrections which are made at prefix-defined errors only, rather than anywhere in a string. The motivation for this concept is that a practical error recovery scheme will be invoked by a parser only at the point at which the parser detects an error, and cannot be expected to make corrections at other points. A model which provides an aim which is more likely to be

achieved in practice would be useful. A prefix-defined error correction is defined by a sequence of edit operations, each of which operates at a prefix-defined error, in an analogous way to a minimum-distance error correction.

*Defn 1.16* A *prefix-defined error correction with n errors* for a string $u$ and a non-empty CFL $L$ over $\Sigma$ is a sentence $v$ of $L$ such that $u \rightarrow v$ *via* $T$ for some sequence of transformations $T$,

$$T = t_1 t_2 \ldots t_n, \qquad t_i = (a_i, b_i) \in \Delta,$$

with strings $w_1, w_2, \ldots, w_{n-1}$ in $\Sigma^*$ where

$u = x_1 a_1 y_1$, $w_1 = x_1 b_1 y_1$, the prefix-defined error in $u$ is after $x_1$ at $a_1$,

$w_i = x_{i+1} a_{i+1} y_{i+1}$, $w_{i+1} = x_{i+1} b_{i+1} y_{i+1}$, the prefix-defined error in $w_i$ is after $x_{i+1}$ at $a_{i+1}$ for $i = 1, \ldots, n-2$,

$w_{n-1} = x_n a_n y_n$, $v = x_n b_n y_n$, the prefix-defined error in $w_{n-1}$ is after $x_n$ at $a_n$.

Both a prefix-defined error correction and a minimum-distance error correction consist of a sequence of transformations

$$u = x_1 a_1 y_1 \rightarrow x_1 b_1 y_1 = x_2 a_2 y_2 \rightarrow x_2 b_2 y_2 = \ldots \; x_n a_n y_n \rightarrow x_n b_n y_n = v \in L.$$

The difference between the two is that in the minimum-distance error correction, the transformation can occur anywhere in the string, whereas in the prefix-defined error correction the transformation occurs at the prefix-defined error, and hence $x_i$ is a prefix of $x_{i+1}$ for $i = 1, \ldots, n-1$, and $x_i$ is in Pre(L) for $i = 1, \ldots, n$.

*Defn 1.17* A prefix-defined error correction with $n$ errors for a string $u$ and a non-empty CFL $L$ over $\Sigma$ is *minimum* if there is no prefix-defined error correction with $n$-1 errors for $u$.

*Lemma 1.1* Every string $u$ has a minimum prefix-defined error correction (not necessarily unique) with respect to a non-empty CFL $L$.

*Proof* Let $T$ be the sequence of error transformations $t_1 t_2 \ldots t_m$, $t_i = (a_i, \varepsilon) \in \Delta$, where

$u = x_1 a_1 y_1$, $w_1 = x_1 y_1$, the prefix-defined error in $u$ is after $x_1$ at $a_1$,

$w_i = x_{i+1} a_{i+1} y_{i+1}$, $w_{i+1} = x_{i+1} y_{i+1}$, the prefix-defined error in $w_i$ is after $x_{i+1}$ at $a_{i+1}$

for $i = 1, \ldots, m\text{-}2$,

$w_{m-1} = x_m a_m y_m$, $v = x_m y_m$, the prefix-defined error in $w_{m-1}$ is after $x_m$ at $a_m$,

and there is no prefix-defined error in $v$ ($v$ may be the empty string). $v$ is in Pre($L$), so there is some suffix (continuation string) $x = b_1 \ldots b_n$ such that $vx$ is in $L$. Let $S$ be the sequence of transformations $s_1 s_2 \ldots s_n$, $s_i = (\varepsilon, b_i) \in \Delta$ for $i = 1, \ldots, n$, where $v \rightarrow vx$ *via* $S$. Then $u \rightarrow vx$ *via* $TS$ and $vx$ is a prefix-defined error correction with $m+n$ errors for $u$. $u$ has at least one prefix-defined error correction, hence it must have a (not necessarily unique) minimum prefix-defined error correction.

If a minimum-distance error correction for a string has $n$ errors, then a minimum prefix-defined error correction has at least $n$ errors. As an example, a prefix-defined error correction with three errors for the incorrect Pascal program of Fig. 1.1 is given by deleting the symbols '`to`' and '`10`' and replacing the symbol '`do`' with a semicolon:

```
1     program example(output);
2     begin
3        n := 1; sum := sum + n
4     end.
```

but there is a minimum (but not unique) prefix-defined error correction with two errors,

replacing 'to' by '+' and 'do' by a semicolon:

```
1       program example(output);
2       begin
3          n := 1 + 10; sum := sum + n
4       end.
```

There is no prefix-defined error correction with one error.

It must be borne in mind that error correction is performed at the level of lexical tokens, the units or symbols for syntax analysis, and not at the level of individual source characters. The human who reads the source program makes use of much more information than is available to a parser or syntax error handling scheme, including not only composition of tokens from source characters, but also layout of the program, context-sensitive information such as types, and logical meaning. So the models of minimum-distance and prefix-defined error correction, although theoretically useful and convenient, are limited in their ability to model the user's intention. In the example given above it would seem that minimum-distance correction is more likely than prefix-defined correction to provide the program that the user intended. This is likely to be true in general, as minimum-distance correction gives the shortest sequence of edit operations, whereas prefix-defined correction gives the sequence of edit operations limited to points at which the input prefix is no longer legal. Users do write programs in which the conceptual error occurs before the first illegal symbol.

## 1.5 ERROR DETECTION

A parser detects an error in its input when it has no legal move on its next input symbol. The problem of error detection (with respect to prefix-defined errors) can be said to be both defined and solved by parsers which possess the correct prefix property.

*Defn 1.18* (Aho and Ullman, 1979) A parser $M$ for a CFL $L$ has the *correct prefix property* if for any string $w$ in $\Sigma^*$ of the form $w = xay$ with $x$ in Pre($L$) and $xa$ not in Pre($L$), when $M$ is started in configuration $(q_0, xay, Z_0)$, $M$ halts in a configuration $(q, ay, \gamma)$ with $\delta(q, a, \gamma) = \varnothing$.

The prefix-defined error in a string can be said to be defined by a parser in the sense that a parser with the correct prefix property will halt before consuming the prefix-defined error symbol. Many parsers, including all LL and LR parsers, possess the correct prefix property. This property is important for both error reporting and error recovery, because a diagnostic message locating the exact incorrect symbol can be output and an error recovery scheme can be invoked at the point of error, before consumption of the incorrect symbol, rather than several symbols later.

Although correct prefix parsers never consume incorrect input symbols, Strong LL(1), SLR(1) and LALR(1) parsers may perform parsing actions before halting (Fischer, Tai and Milton, 1979; Graham, Haley and Joy, 1979). Strong LL(1) parsers may pop the top non-terminal on the parsing stack, in an expansion by the empty production. SLR(1) and LALR(1) parsers may replace a sequence of grammar symbols on the parsing stack by a non-terminal in a reduction move, because the method of construction for the parsing tables uses less lookahead information than the method for canonical LR tables. Additionally, the tables of LR parsers may be compacted so that for a state where the only non-error move is a reduce move, this reduction will be indicated for all input symbols; these are called *default reductions* (Graham, Haley and Joy, 1979). The term *immediate error detection property* is used by Fischer et al (1979) and Sippu (1981) for parsers which halt as soon as the prefix-defined error is the next symbol of the remaining input, without performing any actions. The canonical LL and LR parsers possess this property, but will not be of practical interest to us.

# Chapter 2 Literature Survey

The literature contains a large number of papers on syntax error handling, many of which have appeared since the last comprehensive bibliography (Mattern, 1984). There is therefore a need for an up-to-date survey of the literature. In this chapter we aim to chart the development of work on syntax errors, both chronologically and according to the different approaches used; to evaluate that work; and to provide some context and motivation for the new work to be presented in later chapters. Section 2.1 covers previous surveys and bibliographies. Section 2.2 briefly surveys those papers on parsing, parser generators and computation of minimum distance between two strings that give the technical background necessary for our work on error handling. Section 2.3 surveys papers specifically concerned with language design and Section 2.4, papers concerned with the user interface. Section 2.5 surveys papers on methods for error handling. Finally, in Section 2.6 we compare those methods and pose questions which will be addressed in later chapters.

## 2.1 SURVEYS AND BIBLIOGRAPHIES ON ERROR HANDLING

Many textbooks on compilers (for example Bauer and Eickel, 1974; Aho and Ullman, 1977; Bornat, 1979; Aho, Sethi, and Ullman, 1986; Chapman, 1987; Fischer and LeBlanc, 1988) include chapters or sections on syntax error handling which provide an overview of the topic and an introduction to the literature. Ciesinger (1979) and Mattern (1984) give bibliographies of error handling in compilers. Burgess and James (1981) give a bibliography for LR grammars and parsers which includes error handling for LR parsers.

Sippu (1981) includes a comprehensive survey of syntax error handling, describing and classifying different schemes according to the kinds of techniques used to handle errors. Barnard (1981) summarizes some theoretical results by noting that minimum-distance transformation is not always unique and algorithms for it are slow, and classifies methods for error recovery according to which combination of the operations of insertion and deletion

are performed on the parse stack and the input. Dain (1984) surveys work on practical error recovery schemes for LR parsers. Hammond and Rayward-Smith (1984) review error recovery and repair schemes, emphasising practical current techniques and assessing certain properties. Schemes are classified into three broad areas of local, regional and global recovery, where local recovery is defined to be the editing of one or more input symbols, regional recovery takes place over a phrase of the input which may extend in either direction from the prefix-defined error, and global recovery considers the entire input. They conclude that a successful technique must balance the requirements of automation on the one hand and approximating the programmer's intention, by heuristics or even artificial intelligence techniques, on the other hand. They recommend the use of a simple local error recovery scheme backed up by a regional recovery scheme.

Aho (1980) reviews translator writing systems, noting that automatic generation of good diagnostics and error recovery methods is still wanting, despite research and improvements.

The survey of Hall and Dowling (1980) on string comparison also includes a brief review of error handling during parsing.

## 2.2 PARSING, PARSER GENERATORS AND MINIMUM DISTANCE

### 2.2.1 LR Parsing

LR parsing is defined by Knuth (1965) with an algorithm to parse a wide class of languages, the deterministic context-free languages (DCFLs), in linear time. DeRemer's work on Simple LR (SLR) and Lookahead LR (LALR) grammars (DeRemer, 1971) shows how to construct smaller, practical parsers for a subclass of the DCFLs. Sippu and Soisalon-Soininen (1980) give a formal model of LR(k) parsing which also includes formalization of error recovery and correction. The derivations in the system are reversals of rightmost derivations in a grammar augmented with error productions corresponding to error

recovery actions. Aho and Johnson's survey paper (1974) shows that LR parsing is a practical method for compilers. The method is extended for use with ambiguous grammars by Aho, Johnson and Ullman (1975).

### 2.2.2 General Parsing

Some error handling schemes which perform minimum-distance error correction (Aho and Peterson, 1972; Anderson and Backhouse, 1981; Mauney and Fischer, 1982) require a parser capable of handling a highly ambiguous CFG for which LR parsing is inadequate. The general parsers that are used in these schemes are based on algorithms due to Earley (1970) and Graham, Harrison and Ruzzo (1980). Earley's algorithm parses a sentence of any context-free language; it runs in $O(n^3)$ time for ambiguous grammars, $O(n^2)$ time for unambigous grammars, and $O(n)$ time for LR grammars. Graham, Harrison and Ruzzo give another algorithm of cubic time complexity for parsing a general context-free language.

### 2.2.3 Parser Generators

Johnson (1975) describes the LALR parser-generator *yacc* and its error recovery scheme, based on the scheme of Aho and Johnson (1974). Wetherell and Shannon (1981) present an LR(1) parser generator whose parsers halt as soon as they detect an error. Despite this lack of error handling facilities, it has been used to generate parsers for several command languages such as a file handler and a dynamic debugger. Grune and Jacobs (1988) and Gray (1987) present LL(1) parser generators, each of which incorporates into the generated parsers an automatic error-handling scheme based on that of Roehrich (1980). Koskieimis et al (1988) describe the design of a language processor generator that includes parser generators for LL(1) parsers and SLR(1) and LALR(1) parsers. The LL(1) parsers include an error handling scheme based on the follow sets of Wirth (1976). The LR parsers include an error handling scheme based on that of Druseikis and Ripley (1976).

### 2.2.4 String Comparison and Minimum Distance

Comparison of two strings using a minimum distance measure is required for some models of error recovery, including one of the methods presented in Chapter 3. Hall and Dowling (1980) survey approximate string matching. Kruskal (1983) surveys sequence comparison, analyzing a number of metrics on strings which are all based on the Levenshtein (Hamming) minimum distance measure.

There are several papers on algorithms for the computation of the minimum distance between two strings. Wagner and Fischer (1974) give an *O(mn)* algorithm for this problem, where *m*, *n* are the lengths of the strings, using the standard edit operations of insertion, deletion and substitution. Masek and Paterson (1980) solve a slightly restricted version of the problem with an algorithm of running time *O(nm/log n)*. Lowrance and Wagner (1975) give another *O(mn)* algorithm which also permits the edit operation of transposition. Wong and Chandra (1976) show that *O(mn)* operations are necessary (and sufficient) to compute the distance if the edit operations are restricted to tests of equality. Mathies (1988) presents a parallel algorithm to comput the edit distance which runs in *O(log m log n)* time, using *mn* processors on a concurrent read, concurrent write parallel random access machine.

Ehrenfeucht and Haussler (1986) propose a metric for sequence comparison that emphasizes global similarity over sequential matching at the local level. It can be computed in *O(m+n)* time. This metric does not model simple syntactic errors as well as the Hamming minimum distance metric, but could be used to model errors such as the incorrect placement of a Pascal variable definition block.

## 2.3 LANGUAGE DESIGN

In this section we survey work on programming language design related to the single aspect of occurrence and handling of syntax errors, not the much wider questions of

language design either in general or related to other aspects such as programming languages for particular applications. There are few papers on this topic in comparison with work on the design of error handling schemes. There is some work on the design of languages with respect to the kind and frequency of occurrence of errors (Peterson, 1972; Gannon and Horning, 1975; Wetherell, 1977; Ripley and Druseikis, 1978). A few papers are concerned with various measures of size of CFGs (Ginsburg and Lynch, 1976; Bertsch, 1983). Results in this area are of relevance when considering the size of a particular CFG, which determines the size of its LR parser and also the number of symbols or productions to be considered by an automatic error handling scheme. Finally several authors consider ways of making CFGs more suitable for use in particular error handling schemes (Ghezzi, 1975; Fischer, Tai and Milton, 1979; Pai and Kieburtz, 1979 and 1980; Tai, 1980a and 1980b; Mauney and Fischer, 1981 and 1988).

Gannon and Horning (1975) discuss the design of programming languages from the point of view of improving reliability of programs. Both syntactic and semantic errors are considered and it is shown that design decisions have a significant effect on progam reliability. Peterson (1972) and Wetherell (1977) show that some properties desirable for error correction are undecidable for CFGs, in particular that it is undecidable whether minimum-distance errors and prefix-defined errors coincide. Wetherell uses a probabilistic approach, developing probabilistic measures for the occurrence of errors and the distance to detection, and considering the probabilites that an error will be detected and that a repair is a prefix. He suggests that language designers should use such probabilistic analyses in an attempt to make languages less susceptible to common errors. Ripley and Druseikis (1978) give an analysis of a collection of students' Pascal programs which shows the nature of such common errors for Pascal. They find that most syntax errors are simple, the most common being omission of a single token and substitution of an incorrect token, and that errors occur infrequently. They discuss the relationship between such errors and language constructs.

Ginsburg and Lynch (1976) compare grammars for size complexity, in terms of numbers of grammar symbols and productions. They show that only linear improvement in size can be obtained for the CFGs. Bertsch (1983) defines a size measure of efficiency of a CFG as the sum over all productions of the lengths of righthand sides, and gives optimal forms for expressions in Pascal based on experiments measuring parsing times.

Parsers that possess the property of immediate error detection provide error handling schemes with more information than those that may have performed actions on the parse stack before detecting the erroneous input symbol. Ghezzi (1975) defines a subclass of LL(1) grammars which have parsers with this property. Fischer, Tai and Milton (1979) discuss the differences between Strong LL(1) parsers and LL(1) parsers as constructed by the algorithms given by Aho and Ullman (1972, p. 345 and p. 350). Strong LL(1) parsers have smaller parse tables than LL(1) parsers, but do not possess the immediate error detection property, whereas LL(1) parsers do. Both possess the correct prefix property. Fischer, Tai and Milton present a parsing algorithm for non-nullable Strong LL(1) grammars which does perform immediate error detection. Mauney and Fischer (1981) improve the algorithm so that it parses all Strong LL(1) grammars.

Some error handling schemes require the use of special symbols in the language; a CFL to be handled by such a scheme must be of suitable design. Pai and Kieburtz (1979) define *fiducial* symbols of a language which are used by their error recovery algorithm as tokens which constrain the string following them in a sentential form in particular ways. Tai (1980) discusses *predictors*, which are substrings of a sentence that determine the contents of the parse stack when the symbols in the substring are parsed, and shows how these may be used in error recovery. Mauney and Fischer (1988) refine this notion and that of Pai and Kieburtz to identify particular symbols, parsing ahead to which ensures that a certain class of repairs can be constructed.

## 2.4 THE USER INTERFACE - ERROR REPORTING

Most work on error handling gives some attention to the user interface, but Sippu (1981) in his survey notes that it is curious that none of his references is solely devoted to syntax error reporting. Since then the papers of Brown (1982 and 1983) are devoted solely to syntax error reporting by Pascal compilers. Other later papers surveyed here (Dwyer, 1981; Shneiderman, 1982; Norman, 1983) are concerned with the user interface in a context wider than that of compiling, paying attention to all computer system messages. Horning (1974) makes recommendations for both the content and the form of compiler messages to the user, including a classification of responses to errors. Brown (1982 and 1983) criticizes error messages from the user's point of view, describing the current "state-of-the-art" as "appalling". He identifies the "egotistic" compiler as one which presents information in its own terms rather than in those of the user, and he criticizes messages which say not what is wrong but what the compiler expected to find. On the other hand Kantorowitz and Laor (1986) consider that a list of the expected symbols is readily understood by the programmer; one of their prime requirements for an error handling system is the avoidance of inaccurate messages and the automatic generation of useful messages which are described as simple, honest and reliable. Brown also requires the avoidance of inaccurate messages, and suggests that the development of integrated programming environments and the use of windowing systems should improve the quality of error messages.

Dwyer (1981) uses behavioural theory to develop a user-friendly interface in which a system's response to erroneous input consists of repeating the input, indicating the erroneous symbols and displaying the correct format of the input. This model is particularized for a compiler which attempts to print the grammar production it expects to use. Shneiderman (1982) makes recommendations about computer system messages in general including the development of quality control and guidelines, the use of acceptance tests, and the collection of user performance data. Norman (1983) also stresses the importance of analysing users' performance, particularly the errors that are made, to help in the design of interfaces that

improve performance and reduce the number and effect of errors. He recommends that the state of the system and a list of options should be available to the user.

The compaction of diagnostic messages for optimal use of storage space is discussed by Heaps and Radhakrishnan (1977).

## 2.5 ERROR HANDLING SCHEMES

The majority of papers on syntax errors are concerned with the design, implementation and/or evaluation of specific error handling schemes. We shall first survey work from the nineteen-sixties which pioneered the area and supplied some important ideas for later work. The seventies and eighties have seen the development of the area and the publication of many papers. We attempt to survey that work from the seventies which is of relevance to this thesis, and to give a comprehensive survey of work from 1980 to 1988 (Sippu (1981) gives a comprehensive survey of work up to 1980). We classify the papers as far as possible into three areas, according to whether the methods used for handling errors perform global correction, perform alterations to both the parse stack and the input, or perform alterations only to the input at the point of error. These classifications are similar to those of other authors who use the terms global, regional or local correction or recovery to classify schemes (Sippu, 1981; Hammond and Rayward-Smith, 1984). We prefer to use the above classifications as they are more precise. Schemes which alter the parse stack and/or the input may be considered as performing regional recovery; schemes which alter only the upcoming input may be considered as performing local recovery, according to use of these terms by Hammond and Rayward-Smith.

### 2.5.1 Early Work

Production compilers of the sixties such as that for XPL (McKeeman, Horning and Wortman, 1970) traditionally use a form of error recovery called *panic mode*. In this

method, input symbols are deleted until one of a set of marker symbols, determined by the compiler-writer for the particular language, is met. The stack is then popped until the symbol can be shifted. Panic mode recovery is easy to implement and efficient in operation, but often results in the deletion of several input and stack symbols. Several compiler-writers stress the importance of good error handling in a production compiler and develop other techniques than panic mode (Conway and Maxwell, 1963; Moulton and Muller, 1967; Wirth, 1971; Conway and Wilcox, 1973). The Cornell Computing language (CORC) compiler (Conway and Maxwell, 1963) is claimed to be error-correcting. It is followed by a compiler for a dialect of PL/I (Conway and Wilcox, 1973). The aim for this compiler is to complete translation of every program submitted to it, whether correct or not. The methods used for error recovery are ad-hoc and hand-coded. Moulton and Muller (1967) describe a FORTRAN compiler and stress the importance of good error messages, particularly in a teaching context. Gries (1971) suggests a technique for recovery in a bottom-up parser which either deletes the next input symbol or inserts a string of terminal symbols on the input. The PL360 compiler of Wirth (1968) provides a heuristic solution to the problem of error recovery in a precedence parser. There are two cases in which a precedence parser cannot make a legal move, either when the next input symbol cannot be shifted or when there is no production by which to reduce the symbols on the parse stack. In the first case, an attempt is made to perform a local correction by inserting an input symbol chosen from a list. In the second case, a modification to the parse stack is made, replacing stack symbols by a non-terminal chosen from a table. This action is equivalent to reducing by a special production called an *error production.* The list of input symbols for insertion and the table of error productions are supplied by the compiler-writer.

All the schemes described above are ad-hoc, although the techniques used have since been developed for automation. The first paper on automatic error handling, and the only one to be published in the sixties, is due to Irons (1963). It gives a parsing algorithm that carries all possible parses along and an error recovery method which uses the current stack,

which represents the syntax tree built so far, and the upcoming input to alter the input so that parsing can continue.

### 2.5.2 Schemes that Perform Global Correction

Schemes that perform global correction adopt an approach in which the entire input is taken into account when handling errors, with the aim of producing a minimum-distance or least-cost error correction for an incorrect string. Global correction is effected by adapting a general context-free parsing algorithm for use on the entire input, incorporating error transformations either directly into the parsing algorithm or by means of special productions added to the CFG. This approach contrasts with one in which an efficient parser analyzes the input until an illegal symbol is met, at which point a separate error handling scheme is activated.

Aho and Peterson (1972) model the correction of errors by considering three types of transformation of a string: replacement, insertion or deletion of a single symbol. They give an algorithm based on that of Earley (1970) which parses any input string according to a given CFG using the smallest possible number of transformations required to map the string into a syntactically correct string. Error productions which simulate the above transformations are added to the given CFG, yielding an ambiguous grammar generating all sentences over the input alphabet. This grammar is used to guide the parse and count the number of error transformations made. The algorithm requires $O(n^3)$ time and $O(n^2)$ space, rendering it somewhat impractical for use in a production compiler. The paper asks how a language can be designed so as to maximize the edit distance between correct programs; we have already noted in Chapter 1 that common programming languages have programs at edit distance one. Lyon (1974) also gives an algorithm based on Earley's algorithm which computes the minimum number of error transformations needed to transform an incorrect string into a sentence. The algorithm handles all context-free languages.

Krawczyk (1980) performs global recovery by restricting the set of errors that can be handled to those involving single edit operations on a limited set of input symbols, separators and parenthesis structure symbols. A CFG which models such errors is constructed by adding to the original CFG productions with such symbols deleted or replaced. The resulting language must lie within the class of languages of the chosen parser. The error correction is then performed by the parser itself for this CFG. The LR parsing method is extended to handle the ambiguity which commonly arises from the deletion of parenthesis structure symbols from productions.

Wagner (1974) shows how to compute a minimum-distance correction for a regular language, in linear time.

### 2.5.3 Schemes that Alter the Parsing Stack and the Input

The central problem for an error recovery scheme which is not incorporated in the parser is that of determining how to transform a parser configuration in which there is no legal move into one in which there is at least one legal move, so that parsing can continue. The schemes to be considered in this section use several different techniques to solve this problem, in which the common element is modification of both the parsing stack and the input. The problem is now one of determining how much of the parse stack and input to alter and what the inserted stack and input symbols should be. In effect, these schemes aim to identify and replace a portion or *phrase* of the original input extending to left and right of the symbol at the point of discovery of error; input symbols to the left of the error symbol have already been parsed and are thus represented in some way on the parse stack. Identification of the phrase to be replaced is achieved by use of forward moves (Leinius, 1970; Graham and Rhodes, 1975; Levy, 1975; Sippu and Soisalon-Soininen, 1977; Mickunas and Modry, 1978), error productions (Aho and Johnson, 1974; Poonen, 1977; Fischer and Mauney, 1980), marker symbols (Wirth, 1971; Hartman, 1977; Lewi et al, 1978; Pai and Kieburtz, 1979; Pemberton, 1980; Kantorowitz and Laor, 1986), costs

(Spenke et al, 1984), and combinations of these techniques (Graham, Haley and Joy, 1979; Burke and Fisher, 1982; Boullier and Jourdan, 1987; Corbett, 1985).

## Forward Moves

Leinius (1970) develops an error recovery scheme for a precedence parser which formalizes Wirth's idea (Wirth, 1968) of modifying the parse stack by performing an alternative reduction, and introduces the term *phrase-level recovery* to describe the concept. As explained above, phrase-level recovery aims to identify a phrase of the original input in which the error is supposed to occur, and to recover by replacing that phrase with a suitable non-terminal. Leinius also introduces the notion of a *forward move* to gain information about the upcoming input. In the forward move, the error recovery scheme temporarily passes control back to the parser so that some of the remaining input can be parsed; after the forward move, the recovery scheme uses the information gained from the parse ahead to find the shortest sequence of stack and input symbols, bounded by the appropriate precedence relations, that can be replaced by a unique non-terminal which gives a valid reduction and subsequent legal parser configuration. A similar scheme is indicated for LR parsers, implemented by James (1972) for LALR(1) parsers.

Graham and Rhodes (1975) give a scheme for precedence parsers which, like that of Leinius, parses ahead on the input with a forward move in order to gain information. Before the forward move, a *backward move* is made which attempts to make further reductions on the stack. Grammar symbols are assigned costs for insertion and deletion operations, so that a string of symbols with least cost can be chosen as a modification to the stack. A backward move and forward move is also used by Levy (1975) in a method for any left-to-right parser. Errors are assumed to occur in clusters containing up to a fixed number of errors; the backward move is used to find the position of the last special symbol met, called a *beacon*; the forward move is used to generate legal continuations and to select one.

Sippu and Soisalon-Soininen (1977) formalize and extend Leinius' method for LR parsers, and use it as a basis for the generation of automatic error handling in the compiler-writing system HLP78 (Sippu, 1981; Sippu and Soisalon-Soininen, 1982 and 1983), together with spelling correction and local correction. Raiha et al (1980 and 1983), report on experiences gained in using the HLP78 compiler writing system. Ad-hoc suggestions for modifications to CFGs to give better error recovery and diagnostics are made. Mickunas and Modry (1978) present a scheme for LR parsers which is based on the ideas of Levy and Graham and Rhodes, restarting the parser on a forward move and then attempting repair by insertion of a single symbol. Several forward moves may be considered, as the restart states are all those parser states to which there is a transition after shifting the current input symbol.

## Error Productions

An error production is a production added to a CFG which includes the use of a new terminal **error** in the right-hand side of the production. Aho and Johnson (1974) propose an automatic recovery scheme for LR parsers for which the user adds error productions of the form $A \rightarrow$ **error** to the CFG. The error recovery scheme pops stack states until the top state can shift the **error** symbol. The stack is then reduced by the appropriate error production, and input is discarded until a legal shift symbol is met. This scheme is implemented in the parser-generator *yacc* (Johnson, 1978). Poonen (1977) describes a similar scheme for LR parsers, which also uses an **error** terminal in productions added to the grammar by the compiler-writer. When an error is detected, all states in the stack which can shift **error** are inspected to determine all their legal shift symbols. Input is deleted until one of these is met and the stack is popped until a state which can shift this symbol is on top. This method may delete more stack states, but fewer input symbols, than that of Aho and Johnson. Fischer and Mauney (1980) also use error productions together with local repairs.

## Marker Symbols

Several schemes use distinguished terminals of the CFG as marker symbols for purposes of error recovery, usually to delimit portions of the input within which alterations may be made.

Wirth (1971) presents a Pascal compiler for which he considers good syntax error handling to be particularly important because the language and its compiler are to be used for teaching purposes. The compiler contains a top-down syntax analyzer implemented by the method of recursive descent. The method used for error recovery, a sophisticated development of panic mode recovery employing sets of marker symbols known as *follow sets*, is discussed by Wirth (1976) and Amman (1978). At the point of detection of error, the top-down parser has a goal non-terminal on top of its stack, associated with which is a follow set of terminals. The follow set contains those terminals which are legal input symbols for the current parser configuration, together with those terminals which, although not legal, should not be skipped. Error recovery consists of skipping input until such a legal symbol is met, simultaneously rebuilding the partial syntax tree. The recovery method is implemented by an error procedure with a formal parameter that represents a set of symbols compatible with the current syntax tree. Amman (1978) describes the use of an additional parameter that represents a set of symbols compatible with a rebuilt tree. Each parser procedure must call the error procedure with the appropriate parameters. The error recovery scheme in the IBM Pascal/VS compiler, as reported by Spenke et al (1984), is based on the use of follow sets. The method is made more systematic and suitable for generating directly (automatically) from the CFG by Hartmann (1977), with further improvements by Pemberton (1980). Stirling (1985) gives a formal description of follow sets and algorithms for error recovery using follow sets and error productions. These are productions which contain instances of an error message, rather than the error productions of Aho and Johnson (1974) which contain instances of an error token. Kantorowitz and Laor (1986) use a scheme for LL(1) parsers based on follow sets in which ensuring the usefulness of error

messages generated is of primary importance. An error message consists of a list of legal symbols at the point of discovery of error and an indication of which input symbols are skipped in recovery. Lewi et al (1978) describe a technique for top down and LR parsers which uses marker symbols supplied by the compiler-writer to delimit the start and end of a phrase of the input, and a non-terminal reduction goal for such a phrase.

Pai and Kieburtz (1979 and 1980) also use marker or *fiducial* (trustworthy) symbols to identify a phrase for recovery in an LL(1) parser. The parser scans input until such a symbol is met, when the stack is popped until a sentential suffix beginning with the fiducial symbol can be accepted. Tai (1980) and Mauney and Fischer (1988) discuss similar models for marker symbols.

### Costs

Costs of edit operations, as presented by Wagner and Fischer (1974), are used to obtain a locally least-cost repair in the Graham and Rhodes scheme (1975) described above. Spenke et al (1984) use costs of edit operations together with a *reliability value* in a scheme for LL(1) parsers. The reliability value of an input symbol measures the probability that it was not placed in the input accidentally. Insertion and deletion operations are simulated by popping the stack and discarding input symbols until the parser is restored to a valid configuration. The choice of repair is based on a comparison of the total cost of the edit operations with the reliability of the next actual input symbol to be accepted.

### Combinations

Many schemes use combinations of the above techniques, particulary those schemes which can be described as two-level (Graham, Haley and Joy, 1979; Burke and Fisher, 1982; Boullier and Jourdan, 1987). A two-level scheme has two separate stages for error recovery, the second of which is entered only if the first stage fails to produce an acceptable

recovery action. Graham, Haley and Joy (1979) present a two-level scheme for LR parsers which uses ideas from several previous papers: a forward move, to obtain right context, and costs for edit operations, to guide the choice of a repair to the input, in the first stage; and error productions for phrase-level recovery in the second stage if no first-level repair succeeds. The scheme also uses semantic information in hand-coded recovery procedures. Corbett (1985) formalizes and extends this use of semantic information. A potential repair is tested with a forward move with execution of semantic actions; this approach requires the compiler to be able to undo the effects of semantic actions. Corbett also presents techniques for avoiding the effects of default reductions, and a new panic mode algorithm for LR parsers.

Burke and Fisher (1982) give a two-level scheme for LR parsers which attempts local recovery of a single edit operation first, backing up on the parse stack, followed if necessary by phrase-level recovery. The notion of *scope* is used to control the region in which local recovery is attempted, and to provide the goal for secondary recovery by insertion of a string which will close scope. They build on this work with a scheme with three levels of recovery (Burke and Fisher, 1987). The first level performs simple recovery as before; the second level performs scope recovery which deletes and/or inserts input symbols to close an open scope; the third level performs phrase-level recovery which deletes input before or after the symbol at the point of detection of error. Boullier and Jourdan (1987) give a two-level scheme for table-driven parsers and lexical analyzers. The first level attempts insertions, deletions and replacements at the point of detection of error with the aim of finding a match in a list of corrections. The second stage deletes input up to a *key terminal* or marker symbol, and pops the stack until that symbol can be accepted.

### 2.5.4 Schemes that Alter the Input Alone

The designer of an error recovery scheme may choose to limit the possible transformations of parser configurations by allowing modifications to the unexpended input

only. One of the reasons for this choice is that modifying the stack may require changing semantic information associated with the parsed input, which is difficult to achieve in practice (Gries, 1971). The problems to be solved are now those of determining how much of the unexpended input, from the symbol at the point of discovery of error and possibly extending rightwards, should be altered, and determining an appropriate replacement string or repair. Input symbols to the left of the error symbol have already been parsed and are not eligible for edit operations; neither is the current state of the parser to be altered. Several of the techniques used to identify symbols to be replaced in schemes which alter stack and input are also used here, namely forward moves (Druseikis and Ripley, 1976; Pennello and DeRemer, 1978; Koskiemis et al, 1988), marker symbols (Feycock and Lazarus, 1976; Turner, 1977; Roehrich, 1980; Barnard and Holt, 1982; Chytil and Demner, 1987), and costs (Tai, 1978 and 1980a; Backhouse, 1979; Fischer, Milton and Quiring, 1980; Anderson and Backhouse, 1981; Mauney and Fischer, 1982). The use of error productions is inappropriate, as this technique involves replacing stack and input symbols by a suitable non-terminal.

### Forward Move

Druseikis and Ripley (1976) describe a procedure for adding extra states to an SLR parser to handle errors with a forward move. At the point of detection of error, a special non-SLR parser is activated to parse the upcoming input as far as possible. The initial state of this parser is formed from all those states which can accept the error symbol, called recovery states. When this forward move terminates, the original SLR parser is restarted, using information gained from the forward move to choose a restart state which allows parsing of the actual input to continue. This is purely recovery; no actual repair to the input is constructed. Dain (1981) gives an example which shows that this recovery method can simulate the effect of deleting several symbols from the input. A similar method for LR parsers is described by Pennello and DeRemer (1978), who develop the forward move as a parallel parse by the extra recovery states, rather than a union of recovery states. Their scheme attempts to repair the input by insertion, deletion or replacement of a single input

symbol, using information gained from the forward move. LR parsers generated by the HLP84 processor generator (Koskiemis et al, 1988) also use a forward move by a parser augmented with recovery states. An initial recovery state is constructed from all parser states that have a transition on any of a fixed set of "safe" symbols. On detection of an error, the entire parse stack is deleted and the parser restarts from the initial recovery state.

### Marker Symbols

Feycock and Lazarus (1976) make local corrections in a region of the input bounded by a prior choice of marker symbols. Transformations to the error symbol are tried first and if these do not enable the parser to continue, the input is backed up a symbol at a time. They also suggest the use of semantic information in the choice of repair. Turner (1977) develops a variant of panic mode recovery for recursive descent parsers in which input symbols are skipped until a marker symbol is encountered. Marker symbols can be determined automatically as those symbols for which the current parser procedure is checking; additionally the compiler-writer can hand-tune procedures with extra marker symbols. Roehrich (1980) gives methods for automatic error recovery in LL and LR parsers by insertion and deletion operations only on the input. A valid continuation string is found and input is deleted until an *anchor* symbol is met, that is any input symbol which is contained in the continuation. The appropriate prefix of the continuation string is inserted and parsing continues from the anchor symbol. Although the method is automatic, Roehrich also uses semantic considerations applied to common properties of programming languages to hand-tune the choice of anchor symbols, in order to decrease the probability of what he calls avalanches of spurious errors. A similar approach is taken by Chytil and Demner (1987), differing from that of Roehrich in that input is deleted until the first symbol from a fixed set of *skeletal* symbols is met. Informally, skeletal symbols are chosen so that they delimit the region in which an error can be corrected. The authors note that finding a skeletal set for a programming language may be difficult and show that it is undecidable for a CFL whether a set of symbols is its skeletal set. Barnard and Holt (1981) also use *synchronizing* symbols

to mark phrases in the input.

## Costs

Tai (1978) describes a technique called pattern mapping which is used to choose a local repair to the input. Patterns model the transformation of one string into another and have associated costs. A list of patterns is scanned for a successful match with the upcoming input. The method is implemented for an SLR(1) parser. Tai (1980a) also develops a technique for LL(1) parsers which uses costs of edit operations to choose a locally minimum-distance correction, based on a formal model which assumes that errors occur in clusters. Backhouse (1979) describes a method for choosing a local repair to the input based on costs, and applies it in a recursive descent parser. Anderson et al (1983) assess an implementation of the method and Backhouse (1984) analyses the data flow problems arising from optimization of the parser for practical purposes. Anderson and Backhouse (1981) incorporate the same error handling method into Earley's parsing algorithm. Fischer, Milton and Quiring (1980) use costs to choose a repair in a scheme for LL(1) parsers which uses only the insert operation. They show that most common programming languages can be modified so that they are contained in the subclass of LL(1) grammars which can be parsed and corrected by this method. A similar method for LR(1) parsers is developed by Dion (1982), but the technique is less suitable for LR parsing as the stack does not directly contain the information that guides the choice of repair.

Mauney and Fischer (1982) extend the general CFL parser of Graham, Harrizon and Ruzzo (1980) to perform repair which is least-cost over some region of the input. The parsing algorithm simulates the edit operations of insertion, deletion and replacement and includes a computation of costs. It is only invoked when an error is detected by the usual (LL or LR) parser; after some region has been parsed, the chosen repair can be inserted on the input and control returned.

## 2.6 EVALUATION

What advances have been made since 1980, when Aho remarked that automatic generation of good diagnostics and error recovery methods was still lacking in translator writing systems? Many schemes have been developed since 1980. We consider firstly the issue of quality, that is whether good diagnostics and good error recovery methods have been developed, and secondly the issue of automatic generation.

### 2.6.1 Performance Evaluation and Comparison

The question of quality can only be answered by analysing the performance of schemes on faulty source programs. Many of the schemes have been implemented, and the performance of some of them has been evaluated. In this section we attempt to compare previous evaluations of performance.

A method of evaluation used by many authors is to construct a syntax analyser with error recovery for Pascal and test it on the suite of student programs collected by Ripley and Druseikis (1978). The error recovery achieved is then evaluated according to various criteria. Sippu and Soisalon-Soininen (1983) count the number of missed errors, the number of reports of non-existent errors, and the number of tokens skipped in recovery actions. They also classify a recovery action as *excellent* if it is the same as a "competent programmer" might take, *good* if it is not excellent but has no undesirable side effects, *fair* if it induces one missed error or one non-existent error, and *poor* otherwise. Other authors (Pai and Kieburtz, 1978; Pennello and DeRemer, 1978; Burke and Fisher, 1982; Spenke et al, 1984; Boullier and Jourdan, 1987) use similar criteria, classifying Sippu's *fair* and *poor* actions together as *poor*. Stirling (1985) assesses three schemes based on follow sets using slightly different evaluation criteria: a recovery action is *good* if it produces "the most plausible repair", *marginal* if it produces either a minimum-distance correction which is not the most plausible repair, or a repair which is neither minimum distance nor plausible but has

no effect upon the parsing of subsequent symbols, and *poor* otherwise. Anderson et al (1983) use the same criteria as Stirling to assess the scheme of Anderson and Backhouse (1981).

For the purposes of comparison, we classify as *acceptable* both the excellent and good classifications of Sippu, and the good and marginal classifications of Stirling (although Stirling remarks that marginal repairs are not generally acceptable, because they represent inaccurately diagnosed errors; we judge them to be comparable with the good actions of Sippu because none of these actions have undesirable side effects). Fig. 2.1 shows the percentages of recovery actions performed by the schemes on the Ripley suite that are thus classified as acceptable. The schemes are identified by the names of the authors of the scheme. The reference following gives the paper in which the scheme is presented, except in the case of Wirth's follow set scheme and the IBM Pascal/VS scheme, in which cases the reference is to the paper citing the results of assessment.

| | |
|---|---|
| Stirling (a) (1985) | 66% |
| Stirling (b) (1985) | 66% |
| Sippu and Soisalon-Soininen (1983) | 67% |
| Pennello and DeRemer (1978) | 70% |
| Wirth (Stirling, 1985) | 72% |
| Boullier and Jourdan (1987) | 75% |
| IBM Pascal/VS (Spenke et al, 1984) | 77% |
| Pai and Kieburtz (1978) | 77% |
| Anderson and Backhouse (1981) | 79% |
| Spenke et al (1984) | 91% |
| Burke and Fisher (1982) | 98% |

**Fig. 2.1.** Percentage of recovery actions which are acceptable

It will be seen from the figure that in all but two of the schemes, the percentage of acceptable recovery actions lies between 65% and 80%. The very high percentages reported by Spenke et al and Burke and Fisher may be partly due to hand-tuning of their recovery schemes for Pascal. (Both these schemes employ language-specific data on input symbols.)

According to Ripley and Druseikis' analysis of their suite of Pascal programs, 88% of the syntax errors are simple, single token errors (41% missing tokens, 39% incorrect tokens, 8% extra tokens) which could be corrected by a single edit operation (41% insertions, 39% replacements, 8% deletions). It would not be unreasonable to expect a recovery scheme to handle most cases of simple errors well, and this may be one reason why most of the schemes mentioned above achieve similar standards. It is also possible that schemes which achieve lower standards are not considered to be worth publishing.

Some schemes are not assessed by the above methods, but their performance is demonstrated on one or two example programs. These are not usually representative of the majority of programs. The example Algol program of Graham and Rhodes (1975), which has been used by other authors (Druseikis and Ripley, 1976; Tai, 1978; Graham, Haley and Joy, 1979; Fischer, Milton and Quiring, 1980) as a demonstration of their schemes, contains several artificial errors. The general performance of a scheme cannot fairly be judged from its performance on a single example of this kind.

Finally, it is a value judgement whether a scheme that performs "excellent" or "good" recovery action three-quarters of the time is a good error recovery method; and there are other factors to be taken into consideration when making such a judgement. The end user of the scheme, the programmer, may take into consideration such things as the circumstances in which the scheme is used (for example, whether it is in a traditional batch-mode compiler or a syntax-directed editor), the efficiency of the scheme (particularly in response time), and the relative performances of available schemes. The compiler writer may also take into consideration how easily the scheme can be implemented if the parser is to be coded by hand, or how easily the specifications can be written if a parser-generator is to be used. Some compiler writers may wish to tune a scheme for a particular language; others may prefer a scheme which is language independent and requires no understanding of its method in order to be used.

### 2.6.2 Automatic Generation

We now consider the question of automatic generation of error recovery schemes. Several schemes would appear to be suitable for automatic generation and several have actually been incorporated into automatic parser-generators (Roehrich, 1980; Fischer, Milton and Quiring, 1980; Anderson and Backhouse, 1981; Sippu and Soisalon-Soininen, 1983; Spenke et al, 1984; Boullier and Jourdan, 1987; Gray, 1987; Koskiemis et al, 1988; Grune and Jacobs, 1988; ). There are potential problems with each of these systems: the error recovery scheme cannot be completely automatically generated, there are limitations on the grammar accepted, the generated system is inefficient, or there is insufficient data available on the error handling performance.

Several of these schemes cannot be said to be completely automatically generated, because they require the compiler writer either to have some knowledge of the particular scheme, or to supply some language-specific information as additional input to the parser-generator. Sippu's scheme requires a suitably tailored CFG; Koskiemis et al (1988) report that the user of Sippu's scheme needs a thorough understanding of the error recovery method in order to write a CFG with good error recovery properties. Boullier's scheme requires error messages, correction models and optionally, lists of key terminals and terminals which should not be deleted or inserted. Koskiemis' scheme requires a set of safe symbols. Spenke's scheme requires insertion costs and reliability measures. The schemes of Fischer et al and Anderson et al require edit costs.

The schemes of Grune and Jacobs, Fischer et al, and Gray limit the class of languages which can be handled to the LL(1) CFLs. There is no accessible published performance data for the schemes of Grune and Jacobs, Gray, and Roehrich. The PGS compiler-writing system (Dencker, 1985), in which Roehrich's scheme is used, is reported by Gray as producing parsers that are too slow for production compilers. Gray's own parser-generator DEER, which also incorporates a method based on Roehrich, is carefully tuned to produce

parsers which are fast enough.

### 2.6.3 Questions Arising

In response to Aho's statement quoted above on error handling in translator writing systems, it is still pertinent to ask the question: is it possible to design an error handling scheme for incorporation into a parser-generator, which can be completely automatically generated (in other words, a scheme which will not make additional requirements of the compiler writer) and yet which will still produce good diagnostics and error recovery? Subsidiary questions which arise are what work on language design might help, and how can the user interface be improved? In the rest of this thesis work is presented which addresses these questions. In Chapter 3 we discuss goals for error recovery and consider theoretical issues of language and parser design for error recovery. In Chapter 4 we present two new error recovery schemes for an LR parser-generator and a scheme for generating the diagnostic messages of a parser. In Chapter 5 we evaluate the schemes and compare them with others.

# Chapter 3 Good Error Recovery

In this chapter we examine questions about what constitutes a good error recovery scheme, and what contributions context-free languages, grammars and parsing methods can make towards good error recovery.

## 3.1 CRITERIA FOR ERROR RECOVERY

An error recovery scheme should be assessed on several points, as discussed in Chapter 1. A good scheme is one which is satisfactory for all those points which the user considers important. Several authors discuss criteria for a good scheme. Horning (1974) states that a compiler should never loop or crash and it should attempt to detect and report as many errors as possible. He classifies error handling according to six different types of action by the compiler, the actions in increasing order of desirability being to crash or loop, to produce invalid output, to quit, to recover and continue checking, to repair and continue compilation, to correct. He notes that true correction is far from possible with the current (1972) state of language design and compiler techniques. Roehrich (1980) requires an error handling scheme to detect and report all syntactic errors; to emit for each error a clear message which describes the nature and location of the error and explains the recovery action taken; and to recover and continue parsing. In addition, the scheme should neither enlarge the parser's interface to other compiler modules substantially, nor affect adversely the parsing of correct input. Spenke et al (1984) state desirable properties for an error handling scheme to be used in a compiler-generator which include language independence, efficiency, automatic generation of user-oriented messages, detection of all errors, and introduction of no spurious errors. They note that the last two properties cannot be completely formalized.

We shall set goals for an error recovery scheme in the three general areas of performance, efficiency and ease of use. Performance is concerned with the quality of detection and reporting of errors; efficiency is concerned with space and time requirements;

ease of use is concerned with effort by the compiler writer. There are likely to be trade-offs in these areas, particularly between performance and efficiency.

### 3.1.1 Performance

Considering the area of performance first, minimum-distance error correction seems to be the best goal: it comes closest, of all models, to the concept of altering input to the programmer's intention, and it is completely formalized. However, the best known algorithms (Aho and Peterson, 1972; Anderson and Backhouse, 1981; Mauney and Fischer, 1982) are not sufficiently practical in time and space ($O(n^3)$ and $O(n^2)$ respectively) for production compilers, so it is necessary for some purposes to set a goal which may be attained efficiently.

Programmers normally wish to make their programs syntactically correct as quickly as possible, that is with as few passes through the compiler as possible. The parser should therefore attempt to detect all syntactic errors in the input, although without a formal model for errors such as minimum-distance error correction, it is not in general possible to say what is meant by "all syntactic errors". This implies that the parser should analyse all the input, so that no errors go undetected in deleted portions of the input. Thus there must be some means of restarting the parser from an error configuration. In the case of a simple error which can be corrected by a single edit operation, the parser can be restored to a legal configuration by making that edit operation on the input. This repair will frequently be the most natural in the sense that it appears to be close to the programmer's intention, even though there may be a choice of repair. Deleting or replacing one or more input symbols, rather than inserting symbols, may conflict with the aim of parsing as much as possible of the input, but is often a more natural recovery action. The following program from the Ripley collection of Pascal programs (Ripley and Druseikis, 1978) gives an example of an error where the most natural repair is a deletion.

```
1       program p(input, output);
2       begin  if a < b ;
3          then x := 1
4       end.
```

The semicolon on line 2, which will be detected as an illegal symbol by any correct prefix parser, should be deleted. A correction by insertion only requires the insertion of a statement before the semicolon, precipitating a further error at the symbol 'then' requiring further insertions, indicated in italics:

```
1       program p(input, output);
2       begin  if a < b then x := 1 ;
3          if boolean then x := 1
4       end.
```

In the case of a more complex error which cannot be corrected by a single edit operation, there are many ways of restoring the parser to a legal configuration, in different combinations of alterations to the input and alterations to the parsing stack. We require the parser to recover in the sense that it will proceed to parse some or all of the remaining input. The model of minimum prefix-defined error correction presented in Chapter 1 is used to formalize this requirement: the parser should detect exactly $n$ errors as defined by a minimum prefix-defined error correction of $n$ errors, and no more. However, the same practical objection arises as for minimum-distance correction, namely that the goal cannot be achieved efficiently. This is despite the fact that practical (LL and LR) parsers always detect the prefix-defined error in incorrect input, although they cannot be guaranteed to detect the minimum-distance error for the common programming languages. Any method to achieve minimum prefix-defined error correction would need to parse arbitrarily far ahead on the input, or simulate a series of parallel parses, to ensure that minimum correction would result from a particular choice of repair. The following Pascal examples illustrate this point.

```
1     program p1(output);
2     begin
3        a := [ b + c + d + e]
4     end.
```

```
1     program p2(output);
2     begin
3        a := [ b + c + d + e
4     end..
```

The prefix-defined error in each example is on line 3 after 'a :=' at the symbol '['. Possible repairs are deletion of '[' or insertion of an identifier, but it cannot be determined which repair gives the minimum prefix-defined error correction until the closing bracket ']' or the 'end' symbol is met, after an arbitrarily long expression. Hence a practical method cannot find a minimum prefix-defined error correction for all inputs. This is a theoretical aim against which actual performance can be measured.

The parser can assist programmers in correction of errors by producing error messages that give the place in the input at which an error is detected and state what recovery action is taken. Messages should be directed towards the user, describing the error in terms of what the user has done rather than what happened in the parser, and they should be expressed in natural language and the source language, rather than in numbers which have to be looked up in a table, grammar symbols, internal representation or target language (Horning, 1974; Schneiderman, 1982; Brown, 1983).

### 3.1.2 Efficiency

The second area to be considered is that of efficiency. The parser must remain usable:

the space requirements of the error handling scheme should be satisfiable and the time requirements should be acceptable to the user. There should be no additional time requirements for parsing correct input. Users will accept some delay in parsing incorrect input providing that parsing correct input is not delayed and that the error handling scheme provides them with information which helps them to correct their errors. Theoretical measures of time complexity have some use in evaluating acceptability, and ideally an error handling scheme should have the same complexity as the parser, but if the constants involved are large these measures may not be of much practical significance.

### 3.1.3 Ease of Use

The third area is ease of use by the compiler writer. It is not sufficient to design an efficient error recovery scheme with good performance, if such a scheme is difficult to incorporate into a compiler. The method of choice for producing a parser is to use a parser-generator which produces efficient parsers for a wide class of programming languages: LR parsers are efficient and define the DCFLS, but are difficult to produce by hand and are frequently produced by a software tool such as the LALR(1) parser-generator *yacc* (Johnson, 1975). It follows that an error recovery scheme should also be produced by a software tool. If a scheme is to be incorporated into a parser-generator, it must be capable of automatic generation and independent of source language. To make it capable of completely automatic generation, we require that a compiler writer needs no understanding of the error handling scheme in order to use the parser-generator. In particular, if the scheme is to be incorporated into an existing parser-generator, we require that no specifications additional to the existing ones are needed as input to the parser-generator.

### 3.1.4 Goals

The informal goals for an error handling scheme can be summarized as follows:

(1) to detect all errors in the input.

(2) to parse as much as possible of the input.

(3) to repair simple errors and recover from complex errors.

(4) to generate good error messages.

(5) to have practical requirements in time and space.

(6) to have no effect on the analysis of correct input.

(7) to be capable of automatic generation.

(8) to be capable of incorporation into a practical parser-generator.

## 3.2 CFLS AND ERROR RECOVERY

Are there properties of a CFL which make it "good" for error recovery, and if so, can they be assessed or incorporated into a CFL? In this section properties concerning minimum distance between sentences of a CFL, legal continuations for prefixes of a CFL, and distance between minimum-distance errors and prefix-defined errors are considered, and it is shown what relevance these have for error recovery.

### 3.2.1 Minimum Distance between Sentences

If the sentences of a CFL are far apart according to the minimum-distance measure, then an error recovery scheme is more likely to be able to choose a unique repair for an incorrect string. Aho and Peterson (1972) note that "it is interesting to ask how a programming language can be designed so as to maximize the distance between correct programs". This property is known to be undecidable in general.

*Defn 3.1* (Hopcroft and Ullman, 1966) The *internal distance* $D(L)$ of a CFL $L$ is given by

$$D(L) = \min \{ d(x, y) \mid x, y \in L, x \neq y \} .$$

*Lemma 3.1* (Hopcroft and Ullman, 1966) It is undecidable whether the internal distance of

a CFL is equal to $k$, for each $k \geq 1$.

However, it is easily shown that most programming languages do contain sentences (programs) which are of distance one apart. For example, any language which permits the usual arithmetic expressions contains an infinite number of correct programs in which the only difference is between arithmetic expressions `'a + b'` and `'a - b'`.

*Lemma 3.2* If a CFG $G$ contains (useful) productions

$$A \rightarrow \alpha u \beta$$

$$A \rightarrow \alpha v \beta$$

where $\alpha$, $\beta$ are in $V^*$, $u$, $v$ are in $\Sigma^*$ and $d(u, v) = k$, then there are sentences $x$, $y$ in $L(G)$ with $d(x, y) = k$.

Using Lemma 3.2, the following productions give examples of different syntactic constructs which give rise to an infinite number of programs at distance one, for Pascal, Ada and C.

For Pascal:

*adding-operator* → + | - | **or**

For Ada:

*renaming_declaration* → **package** identifier **renames** name ;
| **task** identifier **renames** name ;

For C:

*statement* → **if** ( *expression* ) *statement* | **while** ( *expression* ) *statement*

### 3.2.2 Legal Continuations

Another desirable requirement for error recovery is a property of CFLs concerning suffixes (legal continuations) of prefixes. Good error recovery is only likely to be achieved if a parser can be put back on the right track after repairing the input. Given an input string *uav* with a prefix-defined error after *u* at *a*, if *a* can be repaired by a single edit operation (*c*, *d*), where *c* is *a* or $\varepsilon$, then there is a prefix *ub* where $uav \rightarrow ubv'$ via (*c*, *d*). The set of strings which can follow the prefix *ub* is the left quotient of the CFL by the language consisting of the single string *ub*. This set will be called the *set of legal continuations* of the prefix. If two different edit operations on *a* yield legal prefixes, are their two sets of legal continuations the same? If they are then it is irrelevant which repair is chosen, as far as parsing the remaining input string is concerned. It is shown below that it is undecidable whether an arbitrary CFL possesses the property that for two legal prefixes differing in their last symbol only, the sets of legal continuations are different. This is of practical interest because it implies that it is undecidable whether a given CFL is such that a better choice than an arbitrary one can be made from a set of different insertion or replacement operations. Before formalizing the result we give an example.

The language *L* of expressions involving the binary operators + and - and the operand *a* may be generated by the LL(1) grammar with productions

$$S \rightarrow a\,A$$

$$A \rightarrow +\,S$$

$$A \rightarrow -\,S$$

$$A \rightarrow \varepsilon$$

The string '*a a* + *u* ', where *u* is any string over {*a*, +, -}, has a prefix-defined error after the first symbol at the second '*a*'. The three potential single symbol repairs are insertion of '+' or '-' and deletion at the prefix-defined error symbol '*a*'. The leftmost derivations for the

transformed input strings up to the string $u$ are as follows:

For input '$a + a + u$' (insertion of '+' before '$a$'),

$$S \Rightarrow aA \Rightarrow a+S \Rightarrow a+aA \Rightarrow a+a+S$$

For input '$a - a + u$' (insertion of '-' before '$a$'),

$$S \Rightarrow aA \Rightarrow a-S \Rightarrow a-aA \Rightarrow a-a+S$$

For input '$a + u$' (deletion of '$a$'),

$$S \Rightarrow aA \Rightarrow a+S$$

In all three cases the legal continuations are strings derivable from $S$ (i.e. all sentences of $L$).

If the language is extended to include array elements as operands, with expressions as legal array indices, it may be generated by an LL(1) grammar with productions

$S \rightarrow a\,B\,A$ $\qquad$ $B \rightarrow [\,S\,]$

$A \rightarrow +\,S$ $\qquad$ $B \rightarrow \varepsilon$

$A \rightarrow -\,S$

$A \rightarrow \varepsilon$

The potential single symbol repairs for the string '$a\ a + u$' now include insertion of the symbol '[' as well as the previous repairs. The leftmost derivations for the transformed input strings up to the string $u$ are now:

For input '$a + a + u$' (insertion of '+' before '$a$'),

$$S \Rightarrow aBA \Rightarrow aA \Rightarrow a+S \Rightarrow a+aBA \Rightarrow a+aA \Rightarrow a+a+S$$

For input '$a$ - $a$ + $u$' (insertion of '- ' before '$a$'),

$$S \Rightarrow aBA \Rightarrow aA \Rightarrow a\text{-}S \Rightarrow a\text{-}aBA \Rightarrow a\text{-}aA \Rightarrow a\text{-}a\ +S$$

For input '$a$ + $u$ ' (deletion of '$a$'),

$$S \Rightarrow aBA \Rightarrow aA \Rightarrow a + S$$

For input '$a$ [ $a$ + $u$' (insertion of '[' before '$a$'),

$$S \Rightarrow aBA \Rightarrow a[\,S\,]A \Rightarrow a[\,aBA\,]A \Rightarrow a[\,aA\,]A \Rightarrow a[\,a + S\,]A$$

In the first three cases, the legal continuations are again the strings derivable from $S$; but in the fourth case the legal continuations are the strings derivable from $S$ ]$A$. Choosing insertion of '[' is in this sense different from choosing one of the other repairs, although each repair allows a parser to consume at least one more input symbol. The above languages are simplifications of expressions in common programming languages, not pathological examples.

*Defn 3.2* For a CFL $L$, the *set of legal continuations* Cont($u$) of a string $u$ in Pre($L$) is the left quotient of $L$ by $u$:

$$\text{Cont}(u) = u \setminus L = \{\ v \mid uv \in L\ \}.$$

*Lemma 3.3* For a CFL $L$ and $a, b$ in $\Sigma$, $a \neq b$, it is undecidable whether Cont($ua$) ≠ Cont($ub$) for all $u$ in Pre($L$) such that $ua$ and $ub$ are in Pre($L$).

*Proof* Proof is by reduction of Post's correspondence problem (PCP), which is known to be undecidable (Post, 1946), to the question: is there a string $u$ in Pre($L$) with $ua$ and $ub$ in Pre($L$) such that Cont($ua$) = Cont($ub$)?

Let $X = x_1, x_2,\ldots,x_n$ and $Y = y_1, y_2,\ldots,y_n$ be two lists of words over a finite alphabet $\Sigma$.

Let $a, b, c_1, c_2,\ldots,c_n$ be new symbols. Let $G$ be the CFG

$$G = ( \{S, S_X, S_Y\}, \Sigma \cup \{a, b, c_1, c_2,\ldots,c_n\}, P, S )$$

where $P$ contains the productions

$$S \rightarrow S_X, \qquad S \rightarrow S_Y$$

and for $1 \leq i \leq n$,

$$S_X \rightarrow c_i S_X x_i, \qquad S_Y \rightarrow c_i S_Y y_i,$$

$$S_X \rightarrow c_i a x_i, \qquad S_Y \rightarrow c_i b y_i.$$

Suppose the instance $(X, Y)$ of PCP has a solution, say $i1, i2,\ldots, im$. Let $u$ be the string $c_{im}\ldots c_{i2}c_{i1}$. Then $u$, $ua$ and $ub$ are in Pre($L(G)$) and

$$\text{Cont}(ua) = \text{Cont}(c_{im}\ldots c_{i2}c_{i1}\, a)$$

$$= x_{i1}x_{i2}\ldots x_{im}$$

$$= y_{i1}y_{i2}\ldots y_{im}$$

$$= \text{Cont}(c_{im}\ldots c_{i2}c_{i1}\, b)$$

$$= \text{Cont}(ub).$$

Conversely, suppose there is a $u$ in Pre($L(G)$) with $ua$ and $ub$ in Pre($L(G)$) and Cont($ua$) = Cont($ub$). Then $u$ must be of the form

$$u = c_{im}\ldots c_{i2}c_{i1} \text{ for some } i1, i2,\ldots,im$$

and $\text{Cont}(ua) = x_{i1}x_{i2}\ldots x_{im}$, $\text{Cont}(ub) = y_{i1}y_{i2}\ldots y_{im}$.

Then *i1*, *i2*,..., *im* is a solution to the instance $(X, Y)$ of PCP.

We now give a sufficient (but not necessary) condition for two insertions or replacements to be equivalent in the above sense, that is whether repaired prefixes which result from applying those edit operations have the same sets of legal continuations. First an equivalence relation is defined on terminals and on edit operations.

*Defn 3.3* $a \sim b$ for terminals $a, b$ of a CFG $G = (N, \Sigma, P, S)$, if the condition

$$A \rightarrow \alpha a \beta \text{ is in } P \text{ if and only if } A \rightarrow \alpha b \beta \text{ is in } P$$

holds for all productions in $P$.

Clearly $\sim$ is reflexive, transitive and symmetric.

*Defn 3.4* $(a, b) \approx (a', c)$ for edit operations $(a, b)$, $(a', c)$ in $\Delta$, where $a$ is not $\varepsilon$, if

(i) $a = a'$, and

(ii) for any prefix-defined error after $u$ at $a$ in a string $uav$, if $ub$ and $uc$ are in Pre($L$), then Cont($ub$) = Cont($uc$).

*Lemma 3.4* If $a, b$ are terminals of a CFG such that $a \sim b$, then Cont($ua$) = Cont($ub$) for any $u$ in $\Sigma^*$.

*Proof* Let $G$ be a CFG $(N, \Sigma, P, S)$ and let $a, b$ be terminals of $G$ with $a \sim b$. A string $v$ is in Cont($ua$) if and only if there is some leftmost derivation

$$S \Rightarrow^+ uav$$

iff $S \Rightarrow^+ wA\gamma \Rightarrow w\alpha a\beta\gamma \Rightarrow^* uav$ for some production $A \rightarrow \alpha a \beta$

and strings $w, \gamma$ where $w\alpha \Rightarrow^* u$ and $\beta\gamma \Rightarrow^* v$

iff $S \Rightarrow^{+} wA\gamma \Rightarrow w\alpha b\beta\gamma \Rightarrow^{*} ubv$

iff $S \Rightarrow^{+} ubv$

iff $v$ is in Cont($ub$).

*Corollary* If $b \sim c$ then $(a, b) \approx (a, c)$ for any $a$ in $\Sigma$, $(a, b)$, $(a, c)$ in $\Delta$.

The converse of Lemma 3.4 is false. A counter-example is the CFG $G = (\{S, A, B\}, \{a, b, c\}, P, S\}$ where $P$ contains the productions

$S \rightarrow aA$ $\quad$ $A \rightarrow cA$ $\quad$ $B \rightarrow Bc$

$S \rightarrow bB$ $\quad$ $A \rightarrow c$ $\quad$ $B \rightarrow c$

$G$ generates the regular language $acc^* \cup bcc^*$. Cont($a$) = Cont($b$) = $cc^*$, and for non-empty $u$, Cont($ua$) = Cont($ub$) = $\emptyset$.

By inspection of the grammar for ISO Pascal (Cooper, 1983), the equivalence classes under ~ which contain more than one terminal are the sets

{ +, - }, { *, /, *div, mod, and* } and { =, <, >, <>, <=, >=, *in* }.

For Ada (United States Dept of Defense, 1981) these classes are

{ +, - }, { *, / } and { =, /=, <, >, <=, >= }.

By the corollary to Lemma 3.4, an insertion of '+' is equivalent to an insertion of '-', for example, for these languages.

### 3.2.3 Distance between Minimum-Distance and Prefix-Defined Errors

It has already been noted (Ch. 1, p. 21) that the first minimum-distance error (uniquely defined as leftmost error in longest prefix correction) may not coincide with the prefix-defined error in an incorrect string. Peterson (1972) has shown that it is undecidable

for an arbitrary DCFL whether it has the property that the minimum distance between minimum-distance errors and prefix-defined errors is bounded by $k$ for each integer $k \geq 1$. We give a necessary condition for an arbitrary CFL to have the property that, for strings at minimum distance one from a sentence of the language, the (longest prefix) minimum-distance error coincides with the prefix-defined error. We then use the condition to show that some common programming languages do not have this property.

*Lemma 3.5* If a CFL $L$ possesses the property that for every string at minimum distance one from a sentence of $L$, the (longest prefix) minimum-distance error coincides with the prefix-defined error, then for any non-empty strings $u$ and $v$ in Pre($L$) with $d(u, v) = 1$, if $w$ is in Cont($u$), then there is a string $x$ in Cont($v$) with $d(w, x) \leq 1$.

*Proof* Let $L$ be a CFL with the given property. Let $u$, $v$ be two non-empty strings in Pre($L$) with $d(u, v) = 1$ and let $w$ be in Cont($u$). Consider the string $vw$. If $vw$ is in $L$ then $w$ satisfies the conditions for $x$ above. If $vw$ is not in $L$, since $v$ is in Pre($L$) the prefix-defined error in $vw$ must lie in $w$. Hence the longest prefix minimum-distance error lies in $w$, and there is a minimum-distance error correction $vx$ for $vw$, with $d(x, w) = 1$.

*Corollary* A CFL $L = L(G)$ does not possess the property that minimum-distance errors and prefix-defined errors coincide if $G$ contains productions $A \rightarrow \alpha$, $A \rightarrow \beta$ and

$$\alpha \Rightarrow^* u\gamma, \quad \beta \Rightarrow^* v\delta \text{ where } d(u, v) = 1, u \neq \varepsilon, v \neq \varepsilon,$$

and there are no strings $w$, $x$ such that

$$\gamma \Rightarrow^* w, \quad \delta \Rightarrow^* x \quad \text{and } d(w, x) \leq 1.$$

The converse to Lemma 3.5 is false. A counter-example is the language $L = \{a, bc\}$. The non-empty prefixes of $L$ are $a$, $b$ and $bc$. Cont($a$) = Cont($bc$) = $\{\varepsilon\}$, Cont($b$) = $\{c\}$, so all the continuation strings for the prefixes are at distance one or zero. But the string $abc$ is an example in which the minimum-distance error does not coincide with the prefix-defined

error. The longest prefix minimum-distance error correction for *abc* is *bc*, locating the minimum-distance error at *a*, but the prefix-defined error is after *a* at *b*.

There are context-free languages for which the minimum-distance error in a string at distance one from some sentence always coincides with the prefix-defined error. An example is the regular language $L$ over $\{a, b\}$ denoted by the regular expression $a^*ba^*$. The sentences of $L$ are all strings over $\{a, b\}$ that contain exactly one *b*. The set of strings at distance one from some sentence of $L$ is the set of all strings containing either zero or two *b*s, denoted by the regular expression $a^* \cup a^*ba^*ba^*$. Any string containing zero *b*s can be written as $a^i$ for some $i \geq 0$, and has longest prefix minimum-distance error correction $a^ib$, locating the minimum-distance error after $a^i$ at the prefix-defined error. Any string containing two *b*s can be written as $a^iba^jba^k$ for some $i, j, k \geq 0$, and has longest prefix minimum-distance error correction $a^iba^ja^k$, locating the minimum-distance error after $a^iba^j$ at the second *b*, also at the prefix-defined error.

The corollary to Lemma 3.5 is used to show that common programming languages do not possess the property that minimum-distance errors and prefix-defined errors coincide. For Pascal, given the productions

*structured-statement* → *repetitive-statement* | *conditional-statement*

there are derivations from the right-hand sides

*repetitive-statement* $\Rightarrow^+$ **while** *Boolean-expression* **do** *statement*

*conditional-statement* $\Rightarrow^+$ **if** *Boolean-expression* **then** *statement* **else** *statement*

Set

$u$ = **while** *expression*

$v$ = **if** *expression*

$\delta$ = **do** *statement*

$\gamma$ = **then** *statement* **else** *statement*

Any string derived from $\gamma$ has minimum distance at least two from any string derived from $\delta$.

A resulting example is the (incorrect) program

```
program p;
begin
   while x then a := b else c := d
end.
```

This string has a minimum-distance correction of one error, replacing 'while' by 'if', but the prefix-defined error is after 'while x' at 'then'.

For Ada, given the productions

*compound_statement* → *loop_statement* | *if_statement*

there are derivations from the right-hand sides

*repetitive-statement* $\Rightarrow^+$ **while** *condition* **loop** *sequence_of_statements* **end loop ;**

*if_statement* $\Rightarrow^+$ **if** *condition* **then** *sequence_of_statements*
**else** *sequence_of_statements* **end if ;**

Set

$u$ = **while** *condition*

$v$ = **if** *condition*

$\gamma$ = **loop** *sequence_of_statements* **end loop ;**

$\delta$ = **then** *sequence_of_statements* **else** *sequence_of_statements* **end if ;**

Any string derived from $\gamma$ has minimum distance at least two from any string derived from

$\delta$. A resulting example is the string

```
procedure p is
begin
   while x then a := b else c := d end if ;
end p;
```

This string has a minimum-distance correction of one error, replacing 'while' by 'if', but the prefix-defined error is after 'while x' at 'then'.

A similar string in C is

```
f() {
   while (x) a = b; else c = d;
}
```

A minimum-distance correction of one error can be obtained by replacing 'while' by 'if', but there is a longest prefix minimum-distance correction of one error, obtained by deleting 'else'. The prefix-defined error is also at 'else', so in this case the minimum-distance error coincides with the prefix-defined error. The productions

statement → **if** ( expression ) statement

statement → **for** ( expression ; expression ; expression ) statement

give the example C string

```
f() {
   if (e1; e2; e3) a = b;
}
```

which has a unique minimum-distance correction at distance one given by replacing 'if' by 'for', but the prefix-defined error is after 'if (e1' at the first semicolon.

## 3.3 CFGS AND ERROR RECOVERY

We now pose a question about CFGs similar to that posed about CFLs in the previous section: are there properties of a CFG which make it "good" for error recovery and can they be assessed or incorporated into a CFG? If there are such properties, then it would be possible to take them into account when making a choice between grammars generating a particular language, to ensure better error recovery.

### 3.3.1 CFGs which Generate the Required Language

Given two CFGs which are known to generate the same CFL, correct prefix parsers for the two grammars will detect error at the same point in the input, as this depends on the language and not the grammar. For the same reason, the choice of grammar does not affect any recovery action based on repairs involving legal symbols or continuation strings. Given an input string $uav$ with a prefix-defined error after $u$ at $a$, the error is detected when the next input symbol is $a$, and the set of legal continuation strings Cont($u$) is given by

$$\text{Cont}(u) = \{w \mid uw \text{ is in } L\}.$$

A repair of a single edit operation may be made to the input by inspection of the set

$$\text{FIRST}_1(\text{Cont}(u)) = \{a \mid w \Rightarrow^* a\alpha, uw \text{ is in } L \text{ and } a \text{ is in } \Sigma\}$$

which is independent of the choice of grammar to generate the language. On the other hand, phrase-level recovery and follow set recovery are based on information gained from productions and are therefore dependent on the choice of grammar.

Sippu and Soisalon-Soininen (1983) cite experimental results which show that the quality of phrase-level recovery in their scheme depends heavily on the form of the productions. For example, if a list of statements in Pascal is generated by the productions

*statement-sequence* → *statement-sequence* ; *statement* | *statement*

then a semicolon missing from between two statements causes the deletion of the whole of

the second statement. Phrase-level recovery action chooses a non-terminal as a reduction goal and isolates a phrase in the input which is to be replaced by that goal; in this example the non-terminal *statement-sequence* is chosen as the reduction goal and the phrase is delimited at its rightmost end by a semicolon or the **end** symbol. If the less natural productions

*statement-sequence* → *statement_list semicolon statement* | *statement*

*semicolon* → ;

are used instead, the non-terminal *semicolon* is chosen as the reduction goal and the phrase in the input to be replaced is the empty phrase between the two statements. In this case phrase-level recovery is equivalent to local recovery, because the form of the productions has been carefully chosen to ensure insertion or replacement of a suitable terminal.

In syntax analysis, the choice of parsing method and the choice of grammar are inter-dependent; an LR(1) parser is constructed from an LR(1) grammar. In a similar way, in error recovery the question of the choice of grammar is either irrelevant or cannot be considered independently of the method of recovery, because for some methods, recovery is unaffected by the grammar, and for others the quality of recovery depends on the interaction between the grammar and the particular method.

For the error recovery methods to be presented in Chapter 4, recovery is based on legal symbols and is therefore independent of the choice of grammar, except for recovery effected by the second stage of Recovery Method 1. The only framework we have for evaluating different grammars with respect to the quality of error recovery is an experimental one.

### 3.3.2 CFGs which Generate a Superset of the Required Language

A different approach to error recovery is to extend the language to be parsed to a superset, by augmenting the generating grammar with special productions, and allow the parser to handle "errors" directly. If all possible errors are to be handled by the parser, the

augmented grammar will be highly ambiguous and a general parsing algorithm such as Earley's algorithm will be required (e.g. Aho and Peterson, 1972). If a practical parsing algorithm is required, the grammar must be suitable for the chosen parser and so limits must be placed on the errors to be handled directly by the parser. In order for this approach to be effective, knowledge of common errors is required. For example, use could be made of the fact that in Pascal, a missing semicolon is a common error (Ripley and Druseikis, 1978), to add to the productions for a list of statements

*statement-sequence* → *statement-sequence* ; *statement* | *statement*

the additional production

*statement-sequence* → *statement-sequence statement*

in which the semicolon acting as the statement separator is missing. (Reduction by this production should cause an appropriate error message to be output.) Disadvantages of this approach are that the augmented grammar may be ambiguous and hence unsuitable for a practical parser; effective use of the method requires language-specific data on common errors; and the method can only handle those errors specified by the grammar, so a further method will be required to handle all other errors.

## 3.4 PARSING METHOD AND ERROR RECOVERY

Are some parsing methods better than others at facilitating error recovery, and what limitations does the choice of parsing method place on the error recovery mechanisms? We shall consider only LL(1) and LR(1) parsing as these are the practical methods of choice.

### 3.4.1 Detection of Errors

A syntax error in the input will be detected at the same point by LL and LR parsers, because they are correct prefix parsers. However, the error is detected at the prefix-defined error which may not be the same as the minimum-distance error, and this affects possible

recovery action. Peterson (1972) shows that for an arbitrary DCFL and fixed positive integer *k*, it is undecidable whether the distance between the minimum-distance error and the prefix-defined error in a string is bounded by *k*. In fact the distance between minimum-distance errors and prefix-defined errors is unbounded for common programming languages. An example for Pascal is:

```
1       program p(output);
2       begin
3           i := lower to upper do sum := sum + i
4       end.
```

The minimum-distance error occurs on line 3 after 'begin' at the symbol 'i', where the minimum-distance correction inserts the symbol 'for', but the prefix-defined error occurs on line 3 after 'i := lower' at the symbol 'to'. An arbitrarily long expression may be used in place of the expression 'lower', increasing the distance between the minimum-distance error and the prefix-defined error by an arbitrary amount. Practical error recovery strategies for LL and LR parsers cannot therefore perform as satisfactorily on some inputs for common languages as do global error recovering parsers, in terms of finding repairs which give minimum-distance error corrections. For the above example, error recovery based on follow sets (Wirth, 1976) amends the program block to

```
i := lower;
sum := sum + i
```

with a repair of distance 3. Error recovery by insertion only (Fischer, Milton and Quiring, 1980) amends the block to

```
i := lower;
for identifier := identifier to upper do sum := sum + i
```

with a repair of distance 5.

### 3.4.2 Information Available at Point of Detection

When a parser detects an error, it is in a configuration $(q, aw, Z\alpha)$ where $d(q, a, Z) = \emptyset$. The information available to an error recovery scheme consists of the contents of the stack $Z\alpha$, the current state $q$, and the remaining input $aw$. In practice, the stack of an LR parser contains a sequence of states and the stack of an LL parser contains a sequence of grammar symbols, and the driving tables for the parsers are also available. For an LR parser, the top state of the stack and the table for that state encode the handles that are current goals for reduction and the positions in those handles. For an LL parser, the top of the stack encodes either the non-terminal by which to expand, giving the production to be used next in a leftmost derivation, or the terminal expected next on the input. For both parsers then, the current state is associated with a list of expected input symbols. This list enables error recovery methods whose aim is to insert a legal string of symbols on the input, possibly including deletions of actual input symbols: that is, error recovery methods which alter the input only. Methods using phrase-level error recovery which alter the stack as well as the input use a variety of means to identify the phrase to be replaced. Methods which use a goal non-terminal for this purpose are limited to a single non-terminal for each stack item by an LL parser; an LR parser gives a choice of non-terminals for each stack state.

An error recovery method must seek to restore the parser configuration to one in which remaining input can be accepted and which is compatible with previously accepted input, as represented on the parsing stack. This problem can be seen as one of altering the incomplete parse tree. At any point in parsing, the frontier of the parse tree gives the sentential form in the current step of the derivation sequence. The LL parsing stack contains those grammar symbols (nodes of the tree) which occur in the sentential form to the right of the non-terminal currently being expanded in the leftmost derivation. The LR parsing stack contains (conceptually) grammar symbols which occur in the sentential form to the left of the non-terminal to be reduced next, and in addition the states encode all rightmost derivation

sequences in reverse which are compatible with the input so far. This information is available to an error recovery scheme if incorporated at the time of parser generation, through the construction of the collection of sets of LR items.

These issues are illustrated by an example of parsers for a language for expressions generated by the grammar

$$G = (\{E, T\}, \{a, +, (, )\}, P, E\} \tag{3.1}$$

where $P$ contains the productions (numbered for use in the LR parse table in Fig. 3.1)

1. $E \rightarrow T$

2. $E \rightarrow E + T$

3. $T \rightarrow a$

4. $T \rightarrow ( E )$

An LR parse table (with default reductions) for this grammar is shown in Fig. 3.1.

| State | ACTION | | | | | GOTO | |
|---|---|---|---|---|---|---|---|
| | *a* | + | ( | ) | $ | *E* | *T* |
| 0 | S3 | | S4 | | | 1 | 2 |
| 1 | | S5 | | | ACC | | |
| 2 | R1 | R1 | R1 | R1 | R1 | | |
| 3 | R3 | R3 | R3 | R3 | R3 | | |
| 4 | S3 | | S4 | | | 6 | 2 |
| 5 | S3 | | S4 | | | | 7 |
| 6 | | S5 | | S5 | | | |
| 7 | R2 | R2 | R2 | R2 | R2 | | |
| 8 | R4 | R4 | R4 | R4 | R4 | | |

**Fig. 3.1.** SLR(1) parse table for grammar (3.1).

Grammar (3.1) is not suitable for top-down parsing because it contains left recursion.

Eliminating the left-recursive production yields the grammar

$$G' = (\ \{E, E', T\},\ \{a, +, (, )\},\ P',\ E\} \tag{3.2}$$

where $P'$ contains the productions

$$E \rightarrow TE' \qquad T \rightarrow a$$

$$E' \rightarrow +TE' \qquad T \rightarrow (\ E\ )$$

$$E' \rightarrow \varepsilon$$

Grammar (3.2) is LL(1) and generates the same language as grammar (3.1). Its LL parse table is shown in Fig. 3.2.

| | $a$ | + | ( | ) | $ |
|---|---|---|---|---|---|
| $E$ | $E \rightarrow TE'$ | | $E \rightarrow TE'$ | | |
| $E'$ | | $E' \rightarrow +TE'$ | | $E' \rightarrow \varepsilon$ | $E' \rightarrow \varepsilon$ |
| $T$ | $T \rightarrow a$ | | $T \rightarrow (\ E\ )$ | | |

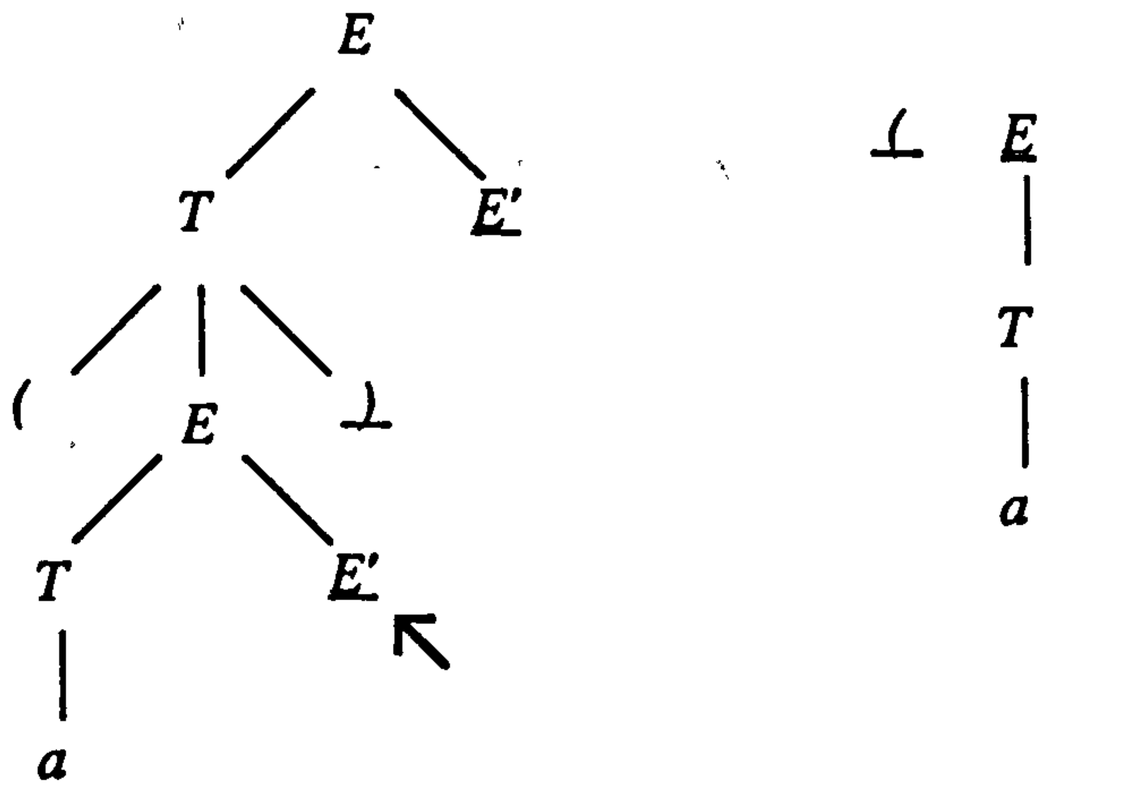
**Fig. 3.2.** LL(1) parse table for grammar (3.2).

We wish to show the information available for the purposes of error recovery, at the point of detection of error, through the different parsing mechanisms of LR and LL parsers. The string '*( a a ...*' is supplied as input to the parsers. The prefix-defined error is after '*( a* ' at the second '*a* '. (The input following the prefix-defined error is not relevant, as it is information available whatever the parsing mechanism.) After consuming the correct prefix '*( a* ', the stack of the LR parser whose table is shown in Fig. 3.1 contains states 0 4 6 (6 on top). The stack of the LL parser whose table is shown in Fig. 3.2 contains the grammar symbols *E' ) E'*. Lists of possible input symbols are readily available to an error recovery method seeking to alter the input, from the tables of the LR and LL parsers. For the LR parser, the top-of-stack state 6 has a table entry which shows that the symbols + and *)* are possible shift symbols. (For an LR parser whose table contains no default reductions, the stack states are 0 4 3 and for state 3, there is a reduction possible on symbols +, *)* or $) For

the LL parser, the top-of-stack grammar symbol has a table entry which shows that + is a possible shift symbol and that a reduction is possible on ) or $ (end of input marker).

Information on non-terminals that are suitable goals for phrase-level recovery is available to an LL parser through its stack contents. The information can be made available to an LR parser from the construction of sets of items at the time of parser generation. In the example, the LL parser's top-of-stack grammar symbol is $E'$. The LR parser's top-of-stack state 6 has kernel items $T \rightarrow ( E \ . \ )$ and $E \rightarrow E \ . + T$ giving goals $T$ and $E$.

Fig. 3.3 shows representations of an incomplete parse tree for the input '( $a$ $a$ ...', as constructed conceptually by the LL parser and the LR parser, at the point of detection of error. The LL parser contains on its stack the symbols $E'$ ) $E'$, indicated by underlined nodes, with the top-of-stack symbol $E'$ indicated by an arrow. The LR parser contains (conceptually) on its stack the grammar symbols ( $E$, underlined in the figure, and has reduced by productions $T \rightarrow a$ and $E \rightarrow T$. In both trees the frontier is '( $a$ ', the input already parsed.




LL parser LR parser

**Fig. 3.3.** Parse trees for input ( $a$ $a$ ...

# Chapter 4 Two Methods for Error Recovery

Two new methods for error recovery are presented in this chapter. Outlines of the methods are given which are independent of parsing method, and then developed for use in conjunction with LR parsers. Schemes are described which implement the methods for incorporation into a well-known LR parser-generator, whose existing method is also described. The differences in approach and the similarites of these schemes to others in the literature is discussed. Finally, a new scheme for use in a scanner-generator and parser-generator is presented, which gives error messages in terms of source input.

## 4.1 NOTATION

Some notation for LR parsers will be required. The states of an LR(1) parsing machine are constructed from sets of LR(1) items of the form $[A \rightarrow \alpha \cdot \ \beta, a]$, where $A \rightarrow \alpha\beta$ is a production of the grammar (augmented with a production $S' \rightarrow S$) and $a$ is a terminal. A configuration of an LR parser is represented by an instantaneous description (ID) $[q_0...q_m, a_j...a_{j+k}]$, where $q_0...q_m$ is the sequence of states on the parsing stack with $q_m$ at the top, and $a_j...a_{j+k}$ is the unexpended input. Moves of an LR(1) parser are given by the ACTION and GOTO transition functions

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

$$\text{GOTO: } Q \times N \rightarrow Q$$

where the relation "move in one step" on parser configurations, denoted by the symbol |-, is defined as follows.

If ACTION$(q_m, a_j)$ = (SHIFT, $q$), then $[q_0...q_m, a_j...a_{j+k}]$ |- $[q_0...q_m q, a_{j+1}...a_{j+k}]$.

If ACTION$(q_m, a_j)$ = (REDUCE, $A \rightarrow a$), $|a| = n$, and GOTO$(q_{m-n}, A) = q$, then

$[q_0 \ldots q_m, a_j \ldots a_{j+k}] \vdash [q_0 \ldots q_{m-n} q, a_j \ldots a_{j+k}]$.

## 4.2 RECOVERY METHOD 1

### 4.2.1 Outline

Recovery Method 1 is a two-level method, that is, it has two stages of operation; the first stage uses the technique of local repair, the second stage, which is activated only if the first stage fails, uses the technique of phrase-level recovery. In the first stage, all single edit operations on the prefix-defined error symbol are considered; if one of these yields a sentence then that is chosen as the correction. Otherwise the second stage of recovery takes place, in which a substring of the original input is replaced by a substring which yields a repaired sentence. The term *repaired sentence* is used to mean a sentence of the language into which the input string is transformed by the error recovery method; similarly the term *repaired string* will be used to mean a string over the input alphabet, not necessarily a sentence of the language, into which the input string is transformed.

1. Let the input string be represented by $uav$ where the prefix-defined error is after $u$ at the symbol $a$.

2. Let $R_1 = \{ ubv' \mid b \text{ in } \Sigma \cup \{\varepsilon\}, uav \rightarrow ubv' \text{ via } (a, b) \text{ or } (\varepsilon, b) \}$

$= \{ ubv \mid b \text{ in } \Sigma \} \cup \{ ubav \mid b \text{ in } \Sigma \} \cup \{ uv \}$.

3. If $R_1 \cap L \neq \varnothing$ then choose any $w$ in $R_1 \cap L$ as the repaired sentence.

4. If $R_1 \cap L = \varnothing$ then find a substring $xy$ of $uav$ and a non-terminal $A$ such that $uav = u'xyv'$ and $S \Rightarrow^+ u'Av' \Rightarrow^+ u'xy'v'$ (possibly $v'$ is the empty string). $u'xy'v'$ is the repaired sentence.

There are two problems to be solved. The first is deciding whether an edit operation yields a sentence, in Step 3. It is not practical to parse the entire remaining input to completion to answer this question, so some approximate answer, or estimate of the suitability of the repair, must be made. The second problem is identifying the substring for replacement in Step 4, the phrase-level recovery.

1. Let the input string be given by $uv$ where $v = v_1...v_n$ for some $v_i$ in $\Sigma$, $i = 1,...,n$, and the prefix-defined error is after $u$ at the symbol $v_1$.

2. Let $R_2 = \{ bv_1...v_\rho \mid b \text{ in } \Sigma \} \cup \{ bv_2... v_\rho \mid b \text{ in } \Sigma \} \cup \{ v_2... v_\rho \}$, where $\rho$ is a fixed integer, $1 \leq \rho \leq n$.

3. If $ux$ is in Pre($L$) for some $x$ in $R_2$, then choose $ux\, v_{\rho+1}...v_n$ as the repaired string.

4. If there is no $x$ in $R_2$ such that $ux$ is in Pre($L$), then find a prefix $u'$ of $u$ and a non-terminal $A$ such that $S \Rightarrow^+ u'A\alpha$ for some $\alpha$ in $V^*$. The repaired string is (or, more strictly, can be derived from) $u'Av_m...v_n$ where $m$ is such that $1 \leq m \leq n$, $S \Rightarrow^+ u'Av_m\beta$ for some $\beta$ in $V^*$ and there is no derivation $S \Rightarrow^+ u'Av_i\gamma$ for $i = 1,..., m\text{-}1$. If there is no such $m$ then the repaired string is $u'A$.

The method outline does not guarantee that a repair leading to a sentence of the language has been found. A repair produced by the primary recovery stage only guarantees that a certain number ($\rho$) of remaining input symbols will be accepted by the parser; secondary recovery either ensures the acceptance of one more symbol, or consumes all the remaining input. It will be necessary to adapt the parsing algorithm to prevent the parser looping in a configuration where there is no more input, but the action entry for the end-of-input marker $

and the current state is ERROR. In this case the error recovery procedure must be called once only. A move of the adapted LR parsing machine is then given by the following:

Let the current configuration of the parser be given by ID $[q_0...q_m, a_j...a_{j+k}]$.

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

If ACTION$(q_m, a_j)$ = (REDUCE, $A \rightarrow a$), $|a| = n$, and GOTO$(q_{m-n}, A) = q$, then

$[q_0...q_m, a_j...a_{j+k}] \vdash [q_0...q_{m-n}q, a_j...a_{j+k}]$.

If ACTION$(q_m, a_j)$ = ACCEPT, halt and accept input.

If ACTION$(q_m, a_j)$ = ERROR, then

1. Call the error recovery procedure.

2. If the configuration of the parser is now $[q_0...q'_m, \$]$ and

   ACTION$(q'_m, \$)$ = ERROR, then halt.

### 4.2.2 Recovery Method 1 for LR Parsing

The method will now be developed as a procedure for use in conjunction with an LR parsing algorithm.

For the first stage of recovery, local repair, suitable symbols for insertion or replacement operations are precisely those symbols for which there is a shift or reduce move from the current state. No other input symbol can give rise to a suffix of the input already parsed. It is an advantage of the LR parsing method that the symbols are readily obtainable from the parsing tables. Having determined which symbols to use in edit operations, a potential repair of a single edit operation can then be assessed by parsing ahead on the input until either a (fixed) number of input symbols have been shifted or the parser accepts, in which case the repair is chosen, or there is no next legal move, in which case the repair is rejected. For the

second stage of recovery, which is invoked if the first stage fails to find any suitable repair, the choice of phrase to be replaced should be guided by the current state of the parse, i.e. a construct for which the parser is amassing a handle. This information is contained in the set of LR items for the current state. The last item to be added to the kernel of the set of items is chosen to provide the goal non-terminal. If its first component is given by

$$[A \rightarrow X_1 ... X_m \cdot \ X_{m+1} ... X_n \ ]$$

then the phrase represented by the item's production right-hand side $X_1 ... X_m X_{m+1} ... X_n$ is to be replaced by the *goal non-terminal* given by the production left-hand side $A$. Input derived from $X_1 ... X_m$ has already been parsed and it is assumed that input resembling a derivation from $X_{m+1} ... X_n$ appears next on the remaining input. $m$ 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 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

$$A \rightarrow X_1 ... X_m \ X_{m+1} ... X_n$$

has been simulated.

Fig. 4.1 gives the scheme developed as a procedure *Recover* in pseudo-Pascal for use by an LR parser.

**procedure** *Recover* ;
**begin**
let the parser configuration when an error is detected be given by the ID

$[q_0 ... q_m, a_j ... a_{j+k}]$;

**for** each symbol $b$ in $\Sigma$ such that ACTION($q_m$, $b$) = (SHIFT, $q$) or

ACTION($q_m$, $b$) = (REDUCE, $A \rightarrow \alpha$) **do**
{ try insertion and replacement by each legal symbol $b$ }
**begin**
{ insertion of $b$ before $a_j$ }
**if** ACTION($q_m$, $b$) = (SHIFT, $q$)
**then** set the parser configuration to $[q_0 ... q_m q, a_j ... a_{j+k}]$
**else** { ACTION($q_m$, $b$) = (REDUCE, $A \rightarrow \alpha$) }
set the parser configuration to $[q_0 ... q_{m-n} q, b a_j ... a_{j+k}]$,

where $|\alpha| = n$ and GOTO($q_{m-n}$, $A$) = $q$ ;
*CheckForwards*;
**if** *CheckOK* **then goto** FINISH;
{ replacement of $a_j$ by $b$ }
**if** ACTION($q_m$, $b$) = (SHIFT, $q$)
**then** set the parser configuration to $[q_0 ... q_m q, a_{j+1} ... a_{j+k}]$
**else** { ACTION($q_m$, $b$) = (REDUCE, $A \rightarrow \alpha$) }
set the parser configuration to $[q_0 ... q_{m-n} q, b a_{j+1} ... a_{j+k}]$,

where $|\alpha| = n$ and GOTO($q_{m-n}$, $A$) = $q$ ;
*CheckForwards*;
**if** *CheckOK* **then goto** FINISH
**end**;
{ try deleting the illegal input symbol $a_j$ }
set the parser configuration to $[q_0 ... q_m, a_{j+1} ... a_{j+k}]$;
*CheckForwards*;
**if** *CheckOK* **then goto** FINISH;
{ secondary recovery }

let the set of items $I_m$ for state $q_m$ have item with first component $A \rightarrow \alpha \cdot \beta$

as the last item to be added to the kernel;

set the parser configuration to $[q_0 \ldots q_{m-n} q, a_j \ldots a_{j+k}]$, where $|\alpha| = n$

and GOTO($q_{m-n}$, $A$) = $q$;

$l := j$;

**while** ACTION($q$, $a_l$) <> SHIFT **and** $l \leq j+k$ **do**

**begin**

**if** ACTION($q$, $a_l$) = (REDUCE, $A \rightarrow \alpha$)

**then** set the parser configuration to $[q_0 \ldots q_{m-n+1} q, a_j \ldots a_{j+k}]$

where $|\alpha| = n$ and GOTO($q_{m-n+1}$, $A$) = $q$ ;

**else begin** { delete next input }

set the parser configuration to $[q_0 \ldots q_{m-n} q, a_{l+1} \ldots a_{j+k}]$;

$l := l+1$

**end**

**end;**

FINISH:

**end;**

**procedure** *CheckForwards*;

**begin**

let the parser configuration be given by the ID $I_0$;

*SymbolsShifted* := 0; *CheckOK* := **true;**

**repeat**

let the parser configuration be given by the ID $[q_0 \ldots q_n, a_l \ldots a_{l+k}]$;

**if** ACTION($q_n$, $a_l$) = ACCEPT

**then** *SymbolsShifted* := $\rho$

**else if** ACTION($q_n$, $a_l$) = (SHIFT, $q$)

**then begin**

set the parser configuration to $[q_0 \ldots q_n q, a_{l+1} \ldots a_{l+k}]$;

*Symbols Shifted* := *SymbolsShifted* + 1

**end**

**else if** ACTION($q_n$, $a_l$) = (REDUCE, $A \rightarrow \alpha$), where $|a| = m$ and

```
                GOTO(q_{n-m}, A) = q,
            then set the parser configuration to [ q_0...q_{n-m}q, a_l ...a_{l+k} ]
        else CheckOK := false
    until SymbolsShifted = ρ or not CheckOK;
    reset the parser configuration to I_0
end;
```

**Fig. 4.1.** Recovery Method 1 for LR parsing.

The procedure *Recover* records the stack and input configuration at the point of detection of error by the parser. It inspects the ACTION table for the current state to determine which input symbols would give rise to a SHIFT or REDUCE move. For each such symbol in turn, the edit operations of insertion before the current illegal symbol, and replacement of the illegal symbol, are simulated by placing the parser in the appropriate configuration. The procedure *CheckForwards* is called to determine whether the edit operation allows the parser to consume a fixed number ($\rho$) of input symbols or accept the input. If so, then control is immediately returned to the main parsing algorithm with the parser set in the edited configuration. If no edit operation meets the criteria, then the parser is set in the configuration obtaining after reduction by the production of the last item in the kernel, followed by repeatedly either reducing if that is indicated by the parse table, or deleting the next input symbol if the table entry is ERROR, until the next input symbol can be shifted.

The procedure *CheckForwards* performs a trial of a repair by parsing ahead on the input, given the repaired configuration of the parser. It makes moves of the parser, recording the number of input symbols consumed in the variable *SymbolsShifted*, until either (a) the parser is in an accepting configuration or has consumed $\rho$ input symbols, in which case the boolean *CheckOK* is set to **true**, or (b) the parser is in an error configuration, in which case *CheckOK* is set to **false**. In either case the parser configuration is reset to what it was when *CheckForwards* was called.

It is necessary to explain the choice of goal non-terminal for the phrase to be replaced in secondary recovery. In the above procedure, the last kernel item is chosen. Choosing a kernel item ensures that at least some of the terminal symbols which the chosen non-terminal may derive have been parsed already. Without any further information, it could be argued that an arbitrary choice from the kernel items should be made. Choosing the last kernel item means that the actions taken in secondary recovery will depend on the way the parser is constructed by the parser-generator. In the case of the parser-generator *yacc*, the kernel items are added according to the order in which the user writes the productions of the grammar for input to *yacc*. Using a top-down approach to writing productions means that the last item in a kernel will have as first component a production for a non-terminal which the user views as a less complex syntactic structure than the ones preceding it. The effect this has on error recovery is to attempt to limit the length of the phrase to be replaced: for example, to replace an expression rather than a statement, or a statement rather than a block. Altering the order in which the productions are written to a random order or a bottom-up order means that a random non-terminal or a more complex non-terminal will be chosen for replacement. However, because of the recursive nature of many language constructs, these notions are not well defined, and the length of phrase replaced will often depend on the actual input rather than the ordering of the grammar productions.

### 4.2.3 Examples

We now give some examples of the actions taken by Recovery Method 1 used with an LALR(1) parser for a small language for simple expressions. The language is defined by the grammar

$$G = ( \{E, T, F\}, \{ id, +, *, (, ) \}, P, E ) \tag{4.1}$$

where $P$ contains the productions (numbered for use with the parse table below)

1. $E \rightarrow T$

2. $E \rightarrow E + T$

3. $T \rightarrow F$

4. $T \rightarrow T * F$

5. $F \rightarrow id$

6. $F \rightarrow ( E )$

An LALR(1) parse table with default reductions (replacing error entries with reductions) for this grammar is shown in Fig. 4.2. 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).

| 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 | | | |

**Fig. 4.2.** LR parse table for grammar (4.1).

The first example demonstrates how local recovery by a single replacement operation is achieved. The string supplied as input to the parser is '*id * ( id id id ) + id + id*'. The error message issued by the implementation of Recovery Method 1 described in Chapter 5 is

```
Line 1: syntax error
id * ( id id id ) + id + id
..........^
'id' replaced by '+'.
```

The parser detects an error when it is in the configuration

[ 0 2 7 5 8, *id id ) + id + id* $ ].

The legal shift symbols for state 8 are *)* and +. Although the symbol * is a valid continuation symbol for the prefix '*id * ( id*', it is not a legal shift symbol because default reductions have been performed, giving viable prefix '*T * ( E*' rather than '*T * ( id*'. Recovery Method 1 first tries insertion of the legal terminals, setting the parser configuration to

[ 0 2 7 5 8 11, *id id ) + id + id* $ ]

which results from shifting the symbol *)*. A forward move on the input is made in order to see whether the repair allows $\rho$ input symbols to be consumed ($\rho$ is 5 in this implementation). The moves of the parser are shown using a slightly different notation for configurations which allows the grammar symbols to be shown on the stack and indicates which productions are used.

| STACK | INPUT | PRODUCTION |
|---|---|---|
| 0 *T* 2 * 7 *(* 5 *E* 8 *)* 11 | *id id ) + id + id* $ | $F \rightarrow ( E )$ |
| 0 *T* 2 * 7 *F* 10 | *id id ) + id + id* $ | $T \rightarrow T * F$ |
| 0 *T* 2 | *id id ) + id + id* $ | $E \rightarrow T$ |
| 0 *E* 1 | *id id ) + id + id* $ | |

In state 1, the action for input symbol *id* is ERROR - the forward move has failed to consume any input symbols. The process is repeated for insertion of the symbol +:

| STACK | INPUT | PRODUCTION |
|---|---|---|
| $0\,T\,2\,*\,7\,(\,5\,E\,8\,+\,6$ | $id\;id\,)\,+\,id\,+\,id\,\$$ | |
| $0\,T\,2\,*\,7\,(\,5\,E\,8\,+\,6\,id\,4$ | $id\,)\,+\,id\,+\,id\,\$$ | $F \rightarrow id$ |
| $0\,T\,2\,*\,7\,(\,5\,E\,8\,+\,6\,F\,3$ | $id\,)\,+\,id\,+\,id\,\$$ | $T \rightarrow F$ |
| $0\,T\,2\,*\,7\,(\,5\,E\,8\,+\,6\,T\,9$ | $id\,)\,+\,id\,+\,id\,\$$ | $E \rightarrow E\,+\,T$ |
| $0\,T\,2\,*\,7\,(\,5\,E\,8$ | $id\,)\,+\,id\,+\,id\,\$$ | |

Again, the forward move terminates in error before sufficient input symbols have been consumed. The recovery method now attempts replacement of the input symbol at the point of error by the legal terminals. The forward move for replacement of *id* by *)* is:

| STACK | INPUT | PRODUCTION |
|---|---|---|
| $0\,T\,2\,*\,7\,(\,5\,E\,8\,)\,11$ | $id\,)\,+\,id\,+\,id\,\$$ | $F \rightarrow (\,E\,)$ |
| $0\,T\,2\,*\,7\,F\,10$ | $id\,)\,+\,id\,+\,id\,\$$ | $T \rightarrow T\,*\,F$ |
| $0\,T\,2$ | $id\,)\,+\,id\,+\,id\,\$$ | $E \rightarrow T$ |
| $0\,E\,1$ | $id\,)\,+\,id\,+\,id\,\$$ | |

This also terminates in error. The next forward move, for replacement of *id* by +, is:

| STACK | INPUT | PRODUCTION |
|---|---|---|
| $0\,T\,2\,*\,7\,(\,5\,E\,8\,+\,6$ | $id\,)\,+\,id\,+\,id\,\$$ | |
| $0\,T\,2\,*\,7\,(\,5\,E\,8\,+\,6\,id\,4$ | $)\,+\,id\,+\,id\,\$$ | $F \rightarrow id$ |
| $0\,T\,2\,*\,7\,(\,5\,E\,8\,+\,6\,F\,3$ | $)\,+\,id\,+\,id\,\$$ | $T \rightarrow F$ |
| $0\,T\,2\,*\,7\,(\,5\,E\,8\,+\,6\,T\,9$ | $)\,+\,id\,+\,id\,\$$ | $E \rightarrow E\,+\,T$ |
| $0\,T\,2\,*\,7\,(\,5\,E\,8$ | $)\,+\,id\,+\,id\,\$$ | |
| $0\,T\,2\,*\,7\,(\,5\,E\,8\,)\,11$ | $+\,id\,+\,id\,\$$ | $F \rightarrow (\,E\,)$ |

| | | |
|---|---|---|
| $0\,T\,2\,*\,7\,F\,10$ | $+\,id\,+\,id\,\$$ | $T \rightarrow T * F$ |
| $0\,T\,2$ | $+\,id\,+\,id\,\$$ | $E \rightarrow T$ |
| $0\,E\,1$ | $+\,id\,+\,id\,\$$ | |
| $0\,E\,1\,+\,6$ | $id\,+\,id\,\$$ | |
| $0\,E\,1\,+\,6\,id\,4$ | $+\,id\,\$$ | $F \rightarrow id$ |
| $0\,E\,1\,+\,6\,F\,3$ | $+\,id\,\$$ | $T \rightarrow F$ |
| $0\,E\,1\,+\,6\,T\,9$ | $+\,id\,\$$ | $E \rightarrow E + T$ |
| $0\,E\,1$ | $+\,id\,\$$ | |
| $0\,E\,1\,+\,6$ | $id\,\$$ | |

At this point, five input symbols have been shifted. The forward move with this local repair has succeeded, so recovery finishes and returns control to the parser with configuration

$$[\,0\,2\,7\,5\,8\,6,\ id\ )\,+\,id\,+\,id\,\$\,].$$

The second example shows secondary recovery by phrase replacement. The string supplied as input to the parser is '*id + ( id id id id ) + id*' . The error message generated is

```
Line 1: syntax error
id + ( id id id id ) + id
..........^
'( id id id id )' replaced by 'F'.
```

The parser detects an error when it is in the configuration

$$[\,0\,1\,6\,5\,8,\ id\ id\ id\ )\,+\,id\,\$\,].$$

The stack corresponds with viable prefix '$E + ( E$' . No single edit operation results in the consumption of enough input symbols during a forward move. Recovery Method 1 resorts to secondary recovery. The kernel items for state 8 are $[E \rightarrow E\,.\,+T\ ]$, $[F \rightarrow (E\,.\,)\ ]$ in that order (ignoring second components as these are not relevant), so $F$ is chosen from the

second item to be the goal non-terminal. Two states, corresponding to the two grammar symbols *( E* to the left of the dot in the right-hand side of the production, are popped from the stack, and the GOTO state for the resulting top of stack state 6 and goal non-terminal *F*, namely state 3, is pushed. The parser configuration is now [ 0 1 6 3, *id id id ) + id* $ ]. State 3 has a reduction on *id* giving configuration [ 0 1 6 9, *id id id ) +id* $ ]. State 9 has a reduction on *id* giving configuration [ 0 1, *id id id ) + id* $ ]. Finally, input symbols are deleted until the action for the current input symbol and top of stack state 1 is SHIFT. The configuration is [ 0 1, *+ id* $ ].

To illustrate the difference in repairs made by secondary recovery when the productions are in a different order, we re-number the productions of the grammar in reverse order:

1. $F \rightarrow ( E )$ (4.2)

2. $F \rightarrow id$

3. $T \rightarrow T * F$

4. $T \rightarrow F$

5. $E \rightarrow E + T$

6. $E \rightarrow T$

The parser for grammar (4.2) has the same parsing table as the first parser (except the production numbers are different), but the kernel items for states are in a different order and hence the non-terminal chosen as goal for secondary recovery is different in some cases. For the example input string *'id + ( id id id id ) + id'* used above, the error message is now

```
Line 1: syntax error
id + ( id id id id ) + id
.........^
'id id id id'  replaced by 'E'.
```

The parser is the same parser as before and detects an error when it is in the same configuration [ 0 1 6 5 8, *id id id )* + *id* $ ]. The kernel items for state 8 are now in reverse order and the last item is [$E \rightarrow E$ . + $T$ ] , so $E$ is chosen as the goal non-terminal. One state is popped from the stack and the GOTO state for the resulting top of stack state 5 and goal non-terminal $E$, namely state 8, is pushed. The parser configuration is now

[ 0 1 6 5 8, *id id id )* + *id* $ ].

No reductions are possible. Input symbols are deleted until the action for the current input symbol and top of stack state 8 is SHIFT. The configuration is [ 0 1 6 5 8, *)* + *id* $ ].

In the first case, secondary recovery has made the repair *'id + F + id'*; in the second case, *'id + ( E ) + id'* . The first grammar has led to choice of $F$ as goal non-terminal and replacement of six actual input symbols, the second to choice of $E$ as goal and replacement of four symbols.

Fig. 4.3 summarizes the differences in recovery resulting from grammars (4.1) and (4.2) for several input strings. The first column shows the actual input string. The second column shows the repair made by recovery based on grammar (4.1) and the third column shows the number of edit operations made. The fourth and fifth column show the repair and the number of edit operations made by recovery based on grammar (4.2). In each of the eight example input strings, the number of minimum-distance errors and the number of prefix-defined errors is two. In the examples shown here, grammar (4.2) produces better error recovery. In general we have no theoretical framework for deciding which of a choice of grammars generating the same language will produce better error recovery by phrase-level recovery methods.

| ACTUAL INPUT | REPAIR MADE (3.1) | NO. OF OPS | REPAIR MADE (3.2) | NO. OF OPS |
|---|---|---|---|---|
| *id + (id id id id) + id* | *id + F + id* | 6 | *id + ( E ) + id* | 4 |
| *id * (id id id id) + id* | *id * F + id* | 6 | *id * ( E ) + id* | 4 |
| *id + (id id id id* | *id + F* | 5 | *id + ( E* | 4 |
| *id * (id id id id* | *id * F* | 5 | *id * ( E* | 4 |
| *id + (id id id id + id + id* | *id + F + id + id* | 5 | *id + ( E + id + id )* | 5 |
| *id * (id id id id + id + id* | *id * F + id + id* | 5 | *id * ( E + id + id )* | 5 |
| *id + (id id id id + id + id)* | *id + F + id + id* | 6 | *id + ( E + id + id )* | 4 |
| *id * (id id id id + id + id)* | *id * F + id + id* | 6 | *id * ( E + id + id )* | 4 |

**Fig. 4.3.** Repairs resulting from different grammars.

### 4.2.4 Termination and Complexity

Termination of the parser with error recovery is assured, for $\rho > 1$, because the error recovery procedure leaves the parser in a configuration in which further symbols of the original input will be consumed. If primary recovery succeeds, at least $\rho$-*1* actual input symbols will be shifted following the repair. If primary recovery fails, then secondary recovery ensures that at least one actual input symbol will be shifted (possibly following some deletions).

The recovery algorithm has complexity *O(n)*. The first loop in the procedure *Recover* invokes the procedure *CheckForwards* at most a constant number (|Σ|) of times, and *CheckForwards* is essentially the LR parsing algorithm and hence *O(n)*. The second loop in *Recover* consists of at most *n* reductions of the parse stack or deletions of input.

## 4.3 RECOVERY METHOD 2

### 4.3.1 Outline

The second method for error recovery consists of a single stage. The idea behind the method is to generate a set of continuation strings for the input already parsed, and choose a string from the set whose minimum distance from the remaining input string is minimum over the set; this string is used to replace the remaining input.

1. Let the input string be represented by $uv$ where the prefix-defined error is after $u$.
2. Let $R_1 = \{ w \mid uw \text{ is in } L\} = \text{Cont}(u)$.
3. Choose $w$ in $R_1$ such that $d(v, w) \leq d(v, w')$ for all $w'$ in $R_1$.
4. The repaired sentence is $uw$.

The problems lie in determining continuation strings, as the set of all such strings may be infinite, and in choosing the one at smallest minimum distance. If there is a means of generating continuation strings in increasing order of length, then the length of the remaining input string and the length of the shortest continuation string can be used to limit consideration of members of the set of continuation strings to members of a finite subset.

*Lemma 4.1* If $v$, $w$ are arbitrary strings with $|v| = n$, $|w| = m$, then

$$|n\text{-}m| \leq d(v, w) \leq \max(n, m).$$

*Proof* A string of length $m$ must be at least $|n\text{-}m|$ edit operations away from a string of length $n$: in the closest case, the strings contain the same symbols; if they are of different lengths then at least $|n\text{-}m|$ insertions are required to transform the shorter string into the longer one. For the upper limit, the worst case is where the strings differ in every symbol.

If $n \geq m$ then $v \rightarrow w$ *via* $T$ where $T$ consists of $m$ replacements followed by $n\text{-}m$ deletions,

so $d(v, w) \leq n$. If $m > n$ then $v \rightarrow w$ *via* $T$ where $T$ consists of $n$ replacements followed by $m$-$n$ insertions, so $d(v, w) \leq m$.

*Lemma 4.2* If $v, w, x$ are arbitrary strings with $|v| = n$, $|w| = m$, $|x| = k$, and $k \geq \max(n, m) + n$, then $d(v, w) \leq d(v, x)$.

*Proof* By Lemma 4.1, $d(v, w) \leq \max(n, m)$ and $|n\text{-}k| \leq d(v, x)$. Also $\max(n, m) + n \leq k$, so $\max(n, m) \leq |n\text{-}k|$. Hence $d(v, w) \leq d(v, x)$.

Using this result, the method outline can be refined as follows.

1. Let the input string be represented by $uv$ where the prefix-defined error is after $u$.
2. Let $w$ be the shortest string in Cont($u$).
3. Let $R_2 = \{ x \mid x \text{ in Cont}(u), |x| < \max(|v|, |w|) + |v| \}$.
4. Choose $x$ in $R_2$ such that $d(v, x) \leq d(v, y)$ for all $y$ in $R_2$.
5. The repaired sentence is $ux$.

The set of continuation strings $R_2$ is finite and can be computed, providing there is a means of computing continuation strings in order of length (the tables of an LR parser give just such a means, in an extension of the previous method used to compute single symbol repairs; the method is given in detail below). We justify the limitation of continuation strings to be considered as repair candidates to the set $R_2$, by claiming that $d(v, x) \leq d(v, y)$ for all $y$ in Cont($u$). If $|y| < \max(|v|, |w|) + |v|$, then by Step 4 $d(v, x) \leq d(v, y)$.

If $|y| \geq \max(|v|, |w|) + |v|$, then by Lemma 4.2, $d(v, x) \leq d(v, y)$. Hence no longer continuation string can give a better repair, in terms of minimum distance from the actual remaining input.

The method is not really practical, as the size of the set of continuation strings $R_2$ is exponential in the length of the remaining input string. The method can be made practical by generating prefixes of a fixed length $\sigma$ of continuation strings and by measuring them against a fixed number $\tau$ of remaining input symbols when making the choice of repair. The chosen prefix of a continuation string will be used to replace a portion of the remaining input rather than the entire string. As with Recovery Method 1, the method does not guarantee to find a repaired sentence. The problems to be solved are which continuation string to choose and how much of the remaining input to replace. It is desirable to obtain the closest match between continuation string and some of the remaining input, but not necessarily 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 a prefix of the fixed amount of input.

1. Let the input string be represented by $uv$ where the prefix-defined error is after $u$ and $v = v_1...v_n$, for $v_i$ in $\Sigma$, $i = 1,...,n$.

2. Let $R_3 = \{ w \mid uw \text{ in } L, |w| < \sigma \} \cup \{ w \mid uw \text{ in Pre}(L), |w| = \sigma \}$, where $\sigma$ is a fixed integer.

3. Choose $x$ in $R_3$ and $m$, $1 \leq m \leq \tau$, such that $d(v_1...v_m, x) \leq d(v_1...v_j, y)$ for all $j$, $1 \leq j \leq \tau$, and all $y$ in $R_3$, where $\tau$ is a fixed integer.

4. The repaired string is $uxv_{m+1}...v_n$.

The method due to Wagner and Fischer (1974) is used to compute the minimum distance $d(u, v)$ between two strings $u = u_1...u_m$ and $v = v_1...v_n$, obtaining a matrix $M$ in which the $(i, j)^{th}$ entry gives the minimum distance between $u_1...u_i$, the prefix of $u$ of length $i$, and

$v_1...v_j$, the prefix of $v$ of length $j$. The algorithm presented by Wagner and Fischer actually computes least cost. It is simplified to compute minimum distance, by letting all edit costs take the value 1, in the procedure shown in Fig. 4.4. The heart of the algorithm computes $d(u_1...u_i , v_1...v_j)$ as the minimum of the three quantities:

(i) $d(u_1...u_{i-1}, v_1...v_j) + 1$, the distance between $u_1...u_{i-1}$ and $v_1...v_j$ plus the insertion of $u_i$

(ii) $d(u_1...u_i , v_1...v_{j-1}) + 1$, the distance between $u_1...u_i$ and $v_1...v_{j-1}$ plus the insertion of $v_j$

(iii) $d(u_1...u_{i-1}, v_1...v_{j-1}) + d(u_i , v_j)$, the distance between $u_1...u_{i-1}$ and $v_1...v_{j-1}$ plus the distance between $u_i$ and $v_j$ (0 if the symbols are the same, 1 if not).

```
procedure MinDist;
begin
for row := 0 to m do M[row, 0] := row;
for col  := 0 to n do M[0, col] := col;
for row := 1 to m do
    for col := 1 to n do
        begin
            a := M[row-1, col] + 1;
            b := M[row, col-1] + 1;
            c := M[row -1, col -1];
            if v[col] <> u[row] then c := c + 1;
            M[row, col] := min(a, b, c)
        end
end;
```

**Fig. 4.4.** Procedure to compute minimum distance.

Using example grammar (4.1) and input string '*id * ( id id id ) + id + id*' again, we take $\tau$ to be 6 and $\sigma$ to be 3. The next $\tau$ 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 )*'. Fig. 4.5 shows the minimum distance matrices for the actual input string '*id id ) + id +*' and the two continuation strings '+ *id* +' and '+ *id )*'.

| | $\varepsilon$ | *id* | *id* | ) | + | *id* | + |
|---|---|---|---|---|---|---|---|
| $\varepsilon$ | 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 |

| | $\varepsilon$ | *id* | *id* | ) | + | *id* | + |
|---|---|---|---|---|---|---|---|
| $\varepsilon$ | 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 |

**Fig. 4.5.** Minimum distance matrices.

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 input, indicated by the smallest entry in the last rows of the two matrices. The repaired string is '*id * ( id + id ) + id + id*'.

We have shown in Chapter 3 that no practical method can guarantee to find a minimum-distance error correction or a minimum prefix-defined 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.

### 4.3.2 Recovery Method 2 for LR Parsing

The problem of generating continuation strings is solved with the use of the LR parsing tables in an extension of the method used to choose a local repair in Recovery Method 1. At 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.

A recovery configuration of an LR(1) parser is represented by an ID $[_R\ q_0...q_m, u\ ]_R$, where $u$ is in $\Sigma^*$, $q_0...q_m$ is the sequence of states on the parsing stack with $q_m$ at the top, and $u$ represents the continuation string. (Brackets $[_R$ and $]_R$ 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 $\vdash_R$, is defined analogously to $\vdash$ on conventional configurations, as follows.

If ACTION($q_m$, $a$) = (SHIFT, $q$), $a$ in $\Sigma$, then $[_R\ q_0...q_m, u\ ]_R \vdash_R [_R\ q_0...q_m q, ua\ ]_R$.

If ACTION($q_m$, $a$) = (REDUCE, $A \rightarrow \alpha$), $|\alpha| = n$, and GOTO($q_{m-n}$, A) = $q$, then

$$[_R\, q_0 \ldots q_m, u\, ]_R \;\vdash_R\; [_R\, q_0 \ldots q_{m-n} q, u\, ]_R.$$

Let $\vdash_R^*$ denote the reflexive and transitive closure of $\vdash_R$. 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 $\Theta_\sigma$ of continuation strings of length up to $s$ symbols is given by

$$\Theta_{RL} = \{\, b_i \ldots b_{i+\sigma-1} \mid b_j \text{ in } \Sigma,\ [_R\, q_0 \ldots q,\ \varepsilon\, ]_R \vdash_R^* [_R\, q_0 \ldots q_n,\ b_i \ldots b_{i+\sigma-1}\, ]_R \,\} \cup$$

$\{\, b_i \ldots b_{i+k} \mid b_j \text{ in } \Sigma,\ 0 \le k < \sigma\text{-}1,\ [_R\, q_0 \ldots q_m,\ \varepsilon\, ]_R \vdash_R^* [_R\, q_0 \ldots q_n,\ b_i \ldots b_{i+k}\, ]_R$ and ACTION($q_n$, \$) = ACCEPT$\}$.

The set $\Theta_\sigma$ of continuation strings of length up to $\sigma$ symbols is generated and the continuation string $x$ whose minimum distance from a prefix of a fixed number $t$ 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 $\Theta_\sigma$ satisfies the following:

Let $\delta = d(x, a_j \ldots a_{j+l}) = \min \{\, d(x, a_j \ldots a_{j+i}) \mid i = 0, \ldots, \tau\text{-}1 \,\}$.

Then for all $y$ in $\Theta_\sigma$, $\delta \le d(y, a_j \ldots a_{j+i})$ for $i = 0, \ldots, \tau\text{-}1$.

Recovery returns control to the parser in the configuration $[q_0 \ldots q_m, x\, a_{j+l+1} \ldots a_{j+k}]$.

Fig. 4.6 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*.

```
procedure Recover;
begin
    let the parser configuration be given by the ID [q_0...q_m, a_j...a_{j+k}];
    RepairString := ε;
    RepairDistance := 2 * τ;
    GenerateRepairs( q_0...q_m, ε);
    set the parser configuration to [q_0...q_m, RepairString a_{j+RepairLength}...a_{j+k}]
end;

procedure GenerateRepairs(Stack, Continuation);
begin
    let the parser stack Stack be denoted by the sequence of states q_0...q_m
    and the continuation string Continuation by the sequence of symbols b_1...b_n;
    if n = σ or ACTION(q_m, $) = ACCEPT
        then for i := 0 to τ-1 do
            if MinDist(Continuation, a_j...a_{j+i}) < RepairDistance
                then begin
                    RepairString := Continuation;
                    RepairDistance := MinDist(Continuation, a_j...a_{j+i});
                    RepairLength := i+1
                end
    else for each symbol b in Σ do
        if ACTION(q_m, b) = (SHIFT, q)
            then GenerateRepairs(q_0...q_m q, b_1...b_n b)
        else if ACTION(q_m, b) = (REDUCE, A → α)
            then GenerateRepairs(q_0...q_{m-p} q, b_1...b_n), where |α| = p and
                GOTO(q_{m-p}, A) = q
end;
```

Fig. 4.6. Recovery Method 2 for LR parsing.

*RepairString* records the best continuation string so far, that is the string of required length $\sigma$ 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.

### 4.3.3 Examples

As an example we use the string '*id * ( id id id ) + id + id*' used above to demonstrate local recovery with Recovery Method 1. The error message issued by the implementation of the recovery method described in Chapter 4 is

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

The parser detects an error when it is in configuration [ 0 2 7 5 8, *id id ) + id + id* $ ].

Generation of repairs commences with recovery configuration $[_R\ 0\ 2\ 7\ 5\ 8, \varepsilon\ ]_R$ . Fig. 4.7 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).

| CONFIGURATION NUMBER | STACK | CONTINUATION STRING |
|---|---|---|
| (1) | 0 *T* 2 * 7 ( 5 *E* 8 | $\varepsilon$ |
| (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* + |

**Fig. 4.7.** Generation of continuation strings.

Fig. 4.8 shows each continuation string generated together with the last row of its minimum distance matrix from the remaining input string '*id id )* + *id* + '.

| | 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 |

**Fig. 4.8.** Continuation strings and minimum distances.

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 prefix-defined error symbol *id* by +. Control is returned to the parser with configuration [ 0 2 7 5 8, + *id )* + *id* + *id* $ ]. In the second example used above to demonstrate secondary recovery by phrase replacement in Recovery Method 1, the string supplied as input to the parser is '*id* + *( id id id id )* + *id* '. The error messages issued by the implementation of Recovery Method 2 are

```
Line 1: syntax error

id + ( id id id id ) + id
..........^    replace 'id' with '+'

Line 2: syntax error

id + ( id id id id ) + id
.............^    replace 'id' with '('

Line 2: syntax error

EOF

^    insert ')'
```

### 4.3.4 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*.

## 4.4 THE LR PARSER-GENERATOR *YACC*

One of the goals established in Chapter 2 for an error recovery scheme was that it should be capable of incorporation into a parser-generator. The parser-generator chosen is *yacc* (Johnson, 1975), for the following reasons. We wished to use an existing parser-generator, to save unnecessary software development. *Yacc* is widely available for use under UNIX systems and the source is available under licence. It has been used for constructing many language translation systems, including the portable C compiler *pcc* (Johnson, 1978), the C++ translator (Stroustrup, 1986), the pattern-matching programming language *awk* (Aho, Kernighan and Weinberg, 1988), and the scanner-generator *lex* (Lesk, 1975). *Yacc* accepts all CFGs and generates LALR(1) parsers (if a CFG is not LALR(1), disambiguating rules are used to construct the parsing tables).

The existing scheme for *yacc* is based on error productions (Aho and Johnson, 1974). If the user does not supply any error productions in addition to the productions of the CFG, no error recovery is built into the generated parser. On detection of an error, the parser issues the message "syntax error" and halts. If the user supplies error productions of the form $A \rightarrow \alpha\ \textit{error}\ \beta$, the method given in Fig. 4.9 is used to effect recovery.

```
procedure Recover;
begin
    let the parser configuration be given by the ID [ q0...qm, aj...aj+k ];
    set the parser configuration to [ q0...qm, error aj...aj+k ];
    i := m; q := qi; n := 0;
    while ACTION(q, error) = ERROR do
        begin
            set the parser configuration to [ q0...qi-1, error aj...aj+k ];
            i := i-1; q := qi
        end;
    while n < 3 do
        begin
            let the parser configuration be given by the ID [ q0...qm, aj...aj+k ];
            if ACTION(qm, aj) = (SHIFT, q) then
                then begin
                    set the parser configuration to [ q0...qmq, aj+1...aj+k ];
                    n := n + 1
                end
            else if ACTION(qm, aj) = (REDUCE, A → a)
                then set the parser configuration to [ q0...qn-pq, aj...aj+k ], where
                |α| = p and GOTO(qn-p, A) = q
            else set the parser configuration to [ q0...qn, aj+1...aj+k ]
        end
end
```

**Fig. 4.9.** Recovery method used by *yacc*.

The actions of the error recovery scheme can be interpreted as follows. The actions in which the ***error*** token is inserted onto the remaining input, and states from the top of the stack are deleted until the top state can shift the ***error*** token, simulate an assumption that an error has occurred in some derivation from a non-terminal, the one with an error production which occurs as an entry symbol for the most recent stack state. The actions in which shift and reduce moves are made as dictated by the parsing tables, with the deletion of any input

symbol for which the table entry is an ERROR entry, simulate deletion from the input of symbols in that derivation. When three input symbols have been consumed, parsing appears to be back on track with the input, so error recovery is assumed to be complete and control is returned to the parser.

The basic error recovery scheme may be altered by the user by use of semantic actions associated with productions, by forcing control to be returned to the parser before three input symbols have been consumed, and by deleting the next input symbol.

The advantages of the method are that it is simple to implement and efficient to run. The disadvantages are that the user has to write error productions which control error recovery to an extent which may not be realized, the method deletes input symbols and stack states silently, and no information about the nature of an error is available. This lack of information leads to poor diagnostic messages.

The portable C compiler gives an example of use of the method. The eight error productions in the input to *yacc* are:

external_def → ***error***

declaration → ***error*** ;

declarator → ***error***

name_list → ***error***

xnfdeclarator → ***error***

init_declarator → ***error***

statement → ***error*** ; | ***error*** }

The effect of these productions in error recovery is that if an error is detected in a statement, input is skipped up to the next following semicolon or } symbol; if an error is detected in a declaration, input is skipped up to the next semicolon; if an error is detected in any other construct, input is skipped up to any symbol which can follow that construct. The usual kind of error message to be output is

```
"file.c", Line 1: syntax error.
```

## 4.5 COMPARISON WITH OTHER METHODS

Recovery Method 1 is a development of a two-stage method for LR parsers due to Graham, Haley and Hoy (1979), consisting of an attempt at a local repair, followed if necessary by phrase-level recovery. There are several differences in the two methods, both in technical details in the two stages and in demands made on the user and the parser. The first stages differ in the criteria used to determine whether a local repair has succeeded. The criterion used in Graham's scheme is whether the cost of the repair is below a fixed threshold, where the cost of a repair is obtained by summing costs associated with the symbols involved in the repair and multiplying by a factor determined by the number of actual input symbols consumed on a forward move; the criterion used in Recovery Method 1 is whether the parser can consume a fixed number of input symbols. The second stages differ in the selection of the goal non-terminal that determines the phrase to replace. Graham's scheme uses an error production, whereas Recovery Method 1 uses the left-hand side of a production in the set of items for the current state. Graham's scheme makes use of semantic information in addition to syntactic information when choosing a repair; Recovery Method 1 does not use semantic information. Graham's scheme requires the parser to delay reductions and the parse tables not to contain default reductions; these requirements are not made of the parser by Recovery Method 1. Graham's scheme requires the user to supply additional information, costs of edit operations and error productions; no extra information is required for Recovery Method 1. Graham's scheme has been implemented in a parser-generator *eyacc* which requires the user to supply hand-coded recovery procedures;

Recovery Method 1 has been implemented in the parser-generator *yacc* with no additional requirements of the user (Dain, 1985).

Recovery Method 2 can be viewed as a development of Recovery Method 1 in which the first stage of recovery is extended to consider repairs of more than a single symbol, removing the need for a second stage. The method used to generate potential repair strings makes repeated use of the single step used for the local repairs of Recovery Method 1. The criterion for choice of repair is best fit with, rather than acceptance of, a certain number of input symbols. The general approach may be likened to that of Roehrich (1980); a major difference is that Recovery Method 2 generates a number of continuations and chooses the repair from these, whereas Roehrich's method generates only one continuation.

## 4.6 AUTOMATIC GENERATION OF ERROR MESSAGES

We now address the question of error messages and show how the automatic generation of messages can be achieved with the methods presented above. We wish to supply the user with information about which symbols are in error and information about symbols which are legal alternatives. The first kind of information is supplied by a message of the form

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

which indicates the prefix-defined error symbol. 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 one of the form

```
Line 32 replaced by

   'while a[1] do begin'
```

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. Messages synthesized by Recovery Method 1 will then take one of the following forms if recovery is effected by stage 1:

```
']' inserted

'do' deleted

']' replaced by '+'
```

and the following form if recovery is effected by stage 2:

```
'procedure f(x:integer):integer' replaced by 'phead'
```

Messages synthesized by Recovery Method 2 take the form

```
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 terminals and non-terminals of the CFG. Our early approach (Dain, 1985) 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 that is 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. A reduction by a production in shift-reduce parsing gives rise to concatenation of the source strings associated with the symbols of the left-hand side of the production; the resulting source string is associated with the right-hand side symbol.

This mechanism means that in error recovery, a repair involving actual input symbols (tokens) and non-terminals obtained by reductions from input symbols can have an associated synthesized message which is expressed in terms of source characters. A repair may also involve insertion of tokens and non-terminals. 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. There is no such possibility for an inserted non-terminal and we revert to the name used in the specification for the parser-generator, which is not necessarily a name familiar to the user. (An inserted non-terminal is only called for in stage 2 of Recovery Method 1.)

Messages are generated at the point of output so there is no need to consider compaction of messages (Heaps and Radhakrishan, 1977).

As examples of error messages, Figs. 4.10 and 4.11 show two Pascal programs from the Ripley collection together with the diagnostic output from Recovery Methods 1 and 2, and from the Berkeley Pascal compiler, whose recovery is based on the work of Graham, Haley and Joy (1979). 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 program in Fig. 4.10 contains a simple error: a semicolon is missing from the end of line 2.

The program in Fig. 4.11 contains a more complex error: the second constant declaration on line 2 contains the expression '`limit+1`', but constant declarations in Pascal may not contain expressions.

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

Recovery Method 1:

```
Line 3: syntax error
   repeat   writeln('  input is:', number)
.....................................^
';' inserted
```

Recovery Method 2:

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

Berkeley Pascal:

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

**Fig. 4.10.** Pascal program containing a simple error.

```
1       program sort(input, output);
2       const limit = 100; limitp1 = limit+1;
3       var q: integer;  begin x := 1 end.
```

Recovery Method 1:

```
Line 3: syntax error
const limit = 100; limitp1 = limit+1;
.....................................^
'const limit = 100; limitp1 = limit+1;' replaced by 'const_decl'
```

Recovery Method 2:

```
Line 3: syntax error
const limit = 100; limitp1 = limit+1;
-----------------------------------^ replace '+' with ';'
-----------------------------------^ insert 'label'
```

Berkeley Pascal:

```
3  const limit = 100; limitp1 = limit+1;
E ----------------------------------^--- Expected ';'
e ----------------------------------^--- Replaced '+' with  a
keyword label
```

**Fig. 4.11.** Pascal program containing a complex error.

Fig. 4.10 shows similar messages generated by all three approaches. Fig. 4.11 shows a slight difference in approach where an expected symbol is listed by the Berkeley compiler. Preference between the two is a matter of taste.

# Chapter 5 Evaluation of Recovery Methods

## 5.1 SATISFACTION OF CRITERIA FOR ERROR HANDLING

Criteria for an error handling scheme were established in Chapter 3. We examine to what extent Recovery Methods 1 and 2 meet these criteria.

The first criterion is that all errors in the input will be detected. Recovery Method 2 satisfies this criterion, but Recovery Method 1 does not, as it might delete some input symbols after repairing the input before returning control to the parser. In this case, it is possible that further undetected errors occur in the input that is not parsed. However, in the case of a simple error of a single erroneous symbol within a small region, the method will repair such an error with a single appropriate edit operation, with no further deletions of input. The criterion will be met if all errors are simple and occur infrequently. (Ripley and Druseikis (1978) show that for student Pascal programs, 87% of errors are simple and occur infrequently.) It will also be met if all complex errors occur infrequently, because in this case any deleted input will not contain errors.

The second criterion is that as much as possible of the input will be parsed. We interpret this strictly as: *all* the input will be parsed. Again, Recovery Method 2 satisfies this criterion, but Recovery Method 1 does not, for the same reason as above, that it might delete input before returning control to the parser. The criterion will be met only if all errors are simple and occur infrequently.

The third criterion is that the error handling scheme will repair simple errors and recover from complex errors. Both methods repair simple errors. Both methods recover from complex errors in the sense that both will terminate and return control to the parser.

The fourth criterion is that good error messages are generated. Both methods use the scheme described in Chapter 4, Section 4.6 to generate error messages. Fig. 5.1 shows examples of error messages generated for a Pascal program containing a single error, an extraneous semicolon on line 2.

```
1  program p(input, output);
2     begin if list[index] < list[loc] ;
3           then x:=1 end.
```

Recovery Method 1:

```
Line 2: syntax error
   begin if list[index] < list[loc] ;
.....................................^
';' deleted.
```

Recovery Method 2:

```
Line 2: syntax error
   begin if list[index] < list[loc] ;
-------------------------------------^ delete ';'
```

**Fig. 5.1.** Error messages for a Pascal program containing one error.

All error messages tell the user exactly which symbol lies at the point of detection of error, and the recovery action taken. They meet the criteria of simplicity, honesty and reliability (Kantorowitz and Laor, 1986; Brown, 1983). For Recovery Method 2, all messages are directed towards the user and expressed in terms which the user understands (Dwyer, 1981). However, for Recovery Method 1, only messages generated if the first stage of local correction succeeds satisfy this criterion. Messages generated by the second stage of phrase-level recovery use grammar names, which are not necessarily meaningful to the user.

Fig. 5.2 shows a Pascal program which illustrates this point. The program contains two minimum-distance errors (two prefix-defined errors) close together and Recovery Method 1 has to use phrase-level correction. The resulting error message uses the non-terminal name *stat* to indicate that some of the input has been replaced by a statement.

```
1  program p(input, output); begin
2     for i := 1 step 1 until listsize - 1 do
3        x := 1 end.
```

Recovery Method 1:

```
Line 2: syntax error
   for i := 1 step 1 until listsize - 1 do
..............^
'for i := 1 step 1 until listsize - 1 do
   x := 1'   replaced by 'stat'
```

Recovery Method 2:

```
Line 2: syntax error
   for i := 1 step 1 until listsize - 1 do
--------------^  replace 'step' with 'to'
----------------------^  replace 'until' with '&'
```

**Fig. 5.2.** Error messages for a Pascal program containing two errors.

The fifth criterion is that requirements in time and space are practical. Both methods satisfy this criterion in the sense that they are 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).

The sixth criterion, that there is no effect on the analysis of correct input, is met by both methods, as the recovery methods are only invoked when the parser detects an error. The parser proceeds exactly as normal on correct input, with no overheads. There is a minor extra space requirement for storage of string representations of the CFG vocabulary symbols.

The seventh criterion, that the error handling scheme can be automatically generated, is met by both recovery methods: no additional specifications are required of the user other than the CFG and semantic actions required by *yacc* itself. In fact fewer specifications are required than formerly, as the user no longer has to supply error productions if error recovery is required.

The eighth criterion, that the error handling scheme can be incorporated into a practical parser-generator, has been met by both methods, by their incorporation into *yacc*.

## 5.2 PERFORMANCE EVALUATION

It is essential to evaluate the performance of an error recovery scheme which is intended for practical use, and to use a representative collection of inputs for that evaluation. Although the parser-generators incorporating Recovery Methods 1 and 2 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 of student Pascal programs (Ripley and Druseikis, 1978) to evaluate performance, so comparisons of our schemes with others will be possible for Pascal. In addition to Pascal and C, we have constructed parsers for various other

languages including C++ (Stroustrup, 1986) and awk (Aho, Kernighan and Weinberger, 1988).

### 5.2.1 Implementation

Recovery Method 1 was implemented in the LR parser-generator *yacc* by the author (Dain, 1985). The implementation was parameterized for the amount of lookahead $\rho$ performed by the recovery method on the input. Recovery Method 2 was implemented in *yacc* by Holloway (1988), as part of an undergraduate project supervised by the author. The implementation was parameterized for the length $\sigma$ of continuation strings generated and the amount of lookahead $\tau$ on the input. The interface between the lexical analyzer and the parser described in Chapter 4, Section 4.6 was implemented for *yacc* and its companion scanner-generator *lex* by Holloway. The implementation of Recovery Method 2 was carried out from a description and informal specification (Dain, 1987), a relevant excerpt from which is contained in Appendix A, and a template parser in C++, contained in Appendix B.

The version of *yacc* incorporating Recovery Method 1 has been distributed to many sites in Europe and the USA, both by the author and by inclusion on a European Unix Users' Group distribution tape. Although the author is not aware of any published details, personal communications report that this version of *yacc* has been used in the construction of many compilers, including ones for Ada, C and occam, "with superior error recovery".

### 5.2.2 Parsers and Inputs

Parsers for C and Pascal were constructed using the two implementations of the parser-generator *yacc* incorporating Recovery Methods 1 and 2, with various values for $\rho$, $\sigma$ and $\tau$. Recovery Method 1 was implemented with values 5 and 10 for $\rho$, the amount of

lookahead on the input. Recovery Method 2 was implemented with the following pairs of values for $\sigma$, the length of strings generated, and $\tau$, 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 (Ripley and Druseikis, 1978) was used as inputs for the Pascal parsers. The database is a reduced sample of original inputs; 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.

### 5.2.3 Method

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, the number of prefix-defined errors, and the number of edit operations performed by the parser with error recovery. A comparison can then be made between the number of minimum-distance or prefix-defined errors, the ideal, with the the number of edit operations, the actual performance. Each input program is associated with an entry in a matrix showing numbers of minimum-distance or prefix-defined errors against numbers of edit operations. For both collections of programs, the Ripley set of Pascal programs (weighted and unweighted) and our set of C programs, for both recovery methods with the various values for $\rho$, $\sigma$ and $\tau$, tables were drawn up giving

total numbers of programs corresponding to each position in the matrix. Results are not available for the set of C programs for Recovery Method 2 with $\sigma$ of 6 and 7, as error recovery took too long to complete with these parameters.

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 (1972). The number of prefix-defined errors is given by the minimum prefix-defined error correction defined in Chapter 1. These are 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.

Fig. 5.1 shows an example of a Pascal program from the Ripley collection and the error messages generated by Recovery Methods 1 and 2. By inspection of the program, it contains one prefix-defined error (and hence one minimum-distance error), an extraneous semicolon on line 2. By inspection of the error messages, Recovery Methods 1 and 2 each make one edit operation. Both recovery methods have attained the ideal. Fig. 5.2 shows a Pascal program from the Ripley collection containing two prefix-defined errors and two minimum-distance errors. Recovery Method 1 makes sixteen edit operations and Recovery Method 2 makes two. Fig. 5.3 shows an example of a Pascal program from the Ripley collection in which the number of minimum-distance errors is not the same as the number of prefix-defined errors; it contains one minimum-distance error and two prefix-defined errors. Recovery Method 1 makes 15 edit operations, Recovery Method 2 makes 5 edit operations.

Fig. 5.4 gives the numbers and percentages of programs in each collection for which the number of prefix-defined errors equals the number of minimum-distance errors. It shows that this is true for the great majority of programs. We expect that the choice between prefix-defined errors and minimum-distance errors as measures will make little difference in results, for these collections of programs. Fig. 5.5 gives the numbers and percentages of programs in each collection which contain only one error, according to both the

prefix-defined error measure and the minimum-distance measure. Most programs contain only one error, whichever of the two measures is used.

```
1  program p(input, output);
2  procedure f(x:integer; var fact:integer):integer;
3     var q:integer; begin x:=1 end; begin end.
```

Recovery Method 1 (15 edit operations):

```
Line 2: syntax error
procedure f(x:integer; var fact:integer):integer;
.....................................^
'procedure f(x:integer; var fact:integer):integer;' replaced
by 'phead'
```

Recovery Method 2 (5 edit operations):

```
Line 2: syntax error
procedure f(x:integer; var fact:integer):integer;
-------------------------------------^ replace ':' with ';'
--------------------------------------^ replace 'integer'
with 'YFORWARD'
Line 3: syntax error
   var q:integer; begin x:=1 end; begin end.
---------------------------------^ delete ';'
-----------------------------------^ delete 'begin'
-------------------------------------^ delete 'end'
```

Minimum-distance error correction (1 error):

```
1  program p(input, output);
2  function f(x:integer; var fact:integer):integer;
3     var q:integer; begin x:=1 end; begin end.
```

Minimum prefix-defined error correction (2 errors):

```
1  program p(input, output);
2  procedure f(x:integer; var fact:integer);
3     var q:integer; begin x:=1 end; begin end.
```

**Fig. 5.3.** Pascal program requiring several edit operations.

| COLLECTION | TOTAL NO. OF PROGRAMS | PROGRAMS WITH SAME NUMBER OF PREFIX-DEFINED AND MIN-DIST ERRORS |
|---|---|---|
| Pascal - unweighted | 121 | 112 *(93%)* |
| Pascal - weighted | 339 | 316 *(93%)* |
| C | 112 | 106 *(95%)* |

**Fig. 5.4.** Numbers of programs with same number of prefix-defined and minimum-distance errors.

| COLLECTION | TOTAL NO. OF PROGRAMS | PROGRAMS WITH ONE PREFIX-DEFINED ERROR | PROGRAMS WITH ONE MIN-DISTANCE ERROR |
|---|---|---|---|
| Pascal - unweighted | 121 | 68 *(56%)* | 72 *(60%)* |
| Pascal - weighted | 339 | 233 *(69%)* | 240 *(71%)* |
| C | 112 | 65 *(58%)* | 67 *(60%)* |

**Fig 5.5.** Numbers of programs containing a single error.

### 5.2.4 Results

The results are shown in Figs. C.1 - C.16 in Appendix C. Figs. C.1 - C.6 are for the Ripley collection of Pascal programs without taking the frequency weights into account, Figs. C.7 - C.12 are for the Ripley collection of Pascal progams with the weights taken into account, and Figs. C.13 - C.16 for the collection of C programs. Each figure consists of

two tables showing the results for a particular recovery method with particular values for $\rho$, $\sigma$ and $\tau$. One of the tables uses minimum-distance errors and the other, prefix-defined errors, as the measure for the number of errors in a program. The columns of each table are indexed by the number of errors, and the rows by 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 57 in column 1, row 1 of the first table of Fig. C.1 shows that 57 programs out of the Pascal collection contained 1 minimum-distance error and required 1 edit operation by Recovery Method 1. The differences between the two tables in each figure are slight, as expected.

### 5.2.5 Analysis

The ideal for error recovery method is to detect the exact number of errors in the input program. This imprecise description is formalized by the number of prefix-defined errors or 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 Figs. C.1 - C.16 occurring on the diagonal. Fig. 5.6 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, with the percentage for prefix-defined errors in brackets. (The differences between the two figures are slight because of the similarities between prefix-defined and minimum-distance errors as measures.)

Although the ideal for 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 Figs. C.1 - C.16?) The answer is given in Fig. 5.7 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, with the percentage for prefix-defined errors in brackets.

| METHOD | PASCAL | PASCAL (WEIGHTED) | C |
|---|---|---|---|
| Recovery Method 1, $\rho = 5$ | *57% (57%)* | *71% (71%)* | *73% (73%)* |
| Recovery Method 1, $\rho = 10$ | *50% (50%)* | *64% (64%)* | *68% (68%)* |
| Recovery Method 2, $\sigma = 4$ | *57% (60%)* | *69% (73%)* | *67% (67%)* |
| Recovery Method 2, $\sigma = 5$ | *55% (57%)* | *64% (65%)* | *65% (67%)* |
| Recovery Method 2, $\sigma = 6$ | *55% (54%)* | *44% (43%)* | - |
| Recovery Method 2, $\sigma = 7$ | *52% (51%)* | *48% (47%)* | - |

**Fig 5.6.** Percentage of programs with number of minimum-distance errors equal to number of edit operations *(prefix-defined errors in brackets)*.

| METHOD | PASCAL | PASCAL (WEIGHTED) | C |
|---|---|---|---|
| Recovery Method 1, $\rho = 5$ | *57% (57%)* | *71% (71%)* | *75% (75%)* |
| Recovery Method 1, $\rho = 10$ | *51% (51%)* | *65% (65%)* | *70% (70%)* |
| Recovery Method 2, $\sigma = 4$ | *74% (77%)* | *83% (87%)* | *76% (77%)* |
| Recovery Method 2, $\sigma = 5$ | *79% (79%)* | *86% (87%)* | *79% (83%)* |
| Recovery Method 2, $\sigma = 6$ | *78% (78%)* | *63% (65%)* | - |
| Recovery Method 2, $\sigma = 7$ | *77% (79%)* | *87% (91%)* | - |

**Fig 5.7.** Percentage of programs with number of minimum-distance errors equal to number of edit operations plus or minus one *(prefix-defined errors in brackets)*.

In addition to the figures on overall performance, it is useful to know how each recovery method performs on programs that contain only a single error, and programs that contain multiple errors. Fig. 5.8 shows the percentage of single-error programs for which the number of minimum-distance errors (prefix-defined errors) equals the number of edit operations, for each collection of programs and for each recovery method. Fig. 5.9 shows the percentage of multiple-error programs for which the number of errors equals the number of edit operations.

## 5.3 DISCUSSION

The first observation to be made from Fig. 5.6 is that the best overall performance is obtained by Recovery Method 1 with lookahead of 5 ( $\rho = 5$): the recovery made achieved the theoretical ideal for 71% of the (weighted) Pascal set and 73% of the C set. The second best overall performance is by Recovery Method 2 with generated repairs of length 4 ( $\sigma = 4$), which achieved the ideal for 69% of the Pascal set and 67% of the C set. This was surprising - Recovery Method 2 was expected to perform better than Recovery Method 1.

| METHOD | PASCAL | PASCAL (WEIGHTED) | C |
|---|---|---|---|
| Recovery Method 1, $\rho = 5$ | 90% *(92%)* | 91% *(92%)* | 97% *(97%)* |
| Recovery Method 1, $\rho = 10$ | 91% *(94%)* | 93% *(94%)* | 98% *(98%)* |
| Recovery Method 2, $\sigma = 4$ | 65% *(69%)* | 77% *(79%)* | 82% *(85%)* |
| Recovery Method 2, $\sigma = 5$ | 67% *(71%)* | 78% *(80%)* | 85% *(88%)* |
| Recovery Method 2, $\sigma = 6$ | 65% *(69%)* | 49% *(50%)* | - |
| Recovery Method 2, $\sigma = 7$ | 64% *(68%)* | 48% *(49%)* | - |

**Fig 5.8.** Percentage of single-error programs with number of minimum-distance errors equal to number of edit operations *(prefix-defined errors in brackets).*

| METHOD | PASCAL | PASCAL (WEIGHTED) | C |
|---|---|---|---|
| Recovery Method 1, $\rho = 5$ | 21% *(20%)* | 29% *(28%)* | 40% *(40%)* |
| Recovery Method 1, $\rho = 10$ | 5% *(5%)* | 5% *(4%)* | 26% *(26%)* |
| Recovery Method 2, $\sigma = 4$ | 45% *(49%)* | 49% *(61%)* | 44% *(43%)* |
| Recovery Method 2, $\sigma = 5$ | 39% *(40%)* | 30% *(33%)* | 36% *(38%)* |
| Recovery Method 2, $\sigma = 6$ | 39% *(34%)* | 32% *(28%)* | - |
| Recovery Method 2, $\sigma = 7$ | 35% *(30%)* | 48% *(43%)* | - |

**Fig 5.9.** Percentage of multiple-error programs with number of minimum-distance errors equal to number of edit operations *(prefix-defined errors in brackets).*

Several other observations can be made. The performance for the Pascal weighted set is better than for the Pascal unweighted set (except in the case of Recovery Method 2 with repairs of length 6 and 7). Both methods give similar performance on Pascal programs compared with C programs. Increasing the length of lookahead for Recovery Method 1 impairs performance. Similarly, increasing the length of generated repair for Recovery Method 2 impairs performance. These last two results were also contrary to expectation.

One of the aims when designing Recovery Method 2 was to improve on the overall performance of Recovery Method 1, by maintaining the good performance on single errors and improving on phrase-level recovery when a single edit operation failed to produce an acceptable repair. It is therefore necessary to seek an explanation for why the overall performance of Recovery Method 1 is better than that of Recovery Method 2. Fig. 5.8 shows that Recovery Method 1 ($\rho = 5$) achieved excellent recovery (number of edit operations equals number of minimum-distance errors) for 91% of the single-error programs in the Pascal collection and 97% of the C collection, compared with Recovery Method 2 ($\sigma = 4$) which achieved excellent recovery for 77% of the Pascal collection and 82% of the C

collection. Fig. 5.9 shows that Recovery Method 1 achieved excellent recovery for 29% of the multiple-error programs in the Pascal collection and 40% of the C collection, compared with Recovery Method 2 which achieved excellent recovery for 49% of the Pascal collection and 44% of the C collection. So Recovery Method 2 does indeed handle multiple errors better than Recovery Method 1, 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 Recovery Methods 1 and 2 find the same single-operation repair, there are also several (ten) for which Recovery Method 1 finds a single-operation repair but Recovery Method 2 finds a repair of two edit operations. For two of these programs it is understandable why Recovery Method 2 has failed to find the single repair. One of them is shown in Fig. 5.10 together with the error messages.

```
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.
```

Recovery Method 1:

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

Recovery Method 2:

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

**Fig. 5.10.** Different repairs for a single-error program.

With generated repairs of length 4 and lookahead on the input of 8 symbols, Recovery Method 2 computes repairs

```
; writeln ( '
```

and

```
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 Recovery Method 1 makes the (arbitrary) choice of the better repair and Recovery Method 2 does not. Not until several symbols later in the actual input is it apparent that the repair made by Recovery Method 2 leads to a further edit operation.

For eight of the ten programs where a single edit operation is found by Recovery Method 1, there is no apparent reason why Recovery Method 2 should not also have found a single repair. An example is given in Fig. 5.11.

```
1  program p(input, output);
2  procedure intlnklst(x, var y : integer);
3     var q: integer; begin x := 1 end; begin end.
```

Recovery Method 1:

```
Line 2: syntax error
procedure intlnklst(x, var y : integer);
.........................^
'var' deleted.
```

Recovery Method 2:

```
Line 2: syntax error
procedure intlnklst(x, var y : integer);
-------------------------^  replace 'var' with 'YID'
-----------------------------^  insert ','
```

**Fig 5.11.** Different repairs for a single-error program.

In theory, Recovery Method 2 should have generated the repair string

```
y : integer )
```

and chosen it as repair, as its distance from the upcoming input is 1. The grammar used by the parser-generators is the same so the failure cannot be attributed to a fault in the grammar. The only other reason the author can surmise is an undetected error or errors in the software implementation of the method. There is therefore no satisfactory explanation for the better performance of Recovery Method 1 overall. If Recovery Method 2 had chosen the same single-operation repairs as Recovery Method 1 for the eight single-error programs for which its choice of repair in practice is not understood, it would have achieved excellent recovery for 88% of single-error Pascal programs and 77% of all Pascal programs (weighted), a better performance overall than Recovery Method 1.

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. Figs. 5.8 and 5.9 show that both recovery methods perform better on single-error programs than on multiple-error programs. For example, Recovery Method 1 ($\rho = 5$) achieved excellent recovery for 90% of single-error programs, but only 21% of multiple-error programs; Recovery Method 2 ($\sigma = 4$) achieved excellent recovery for 65% of single-error programs and 45% of multiple-error programs. Hence the weighting improves the performance figures.

Performance for the Pascal weighted set is similar to performance for the C set. Recovery Method 1 ($\rho = 5$) achieved excellent recovery for 73% of C programs and 71% of Pascal programs; Recovery Method 2 ($\sigma = 4$) achieved excellent recovery for 69% of the Pascal set and 67% of the C set. The difference of 2% is not significant given the number of programs in each set.

When the amount of lookahead $\rho$ on the input which Recovery Method 1 performs is increased from 5 to 10, performance is degraded, from excellent recovery on 71% of Pascal programs down to 64%, and from 73% down to 68% for C programs. This was contrary to expectation: it was thought that increasing the amount of lookahead on which a repair is assessed would increase the likelihood of choosing a good repair. The degradation can be explained by clustering of errors. Performance for single-error programs is actually improved slightly by increasing lookahead, from excellent recovery for 91% up to 93% for Pascal and from 97% up to 98% for C. It is on multiple-error programs that performance is degraded, from excellent recovery for 29% down to 5% for Pascal and from 40% down to 26% for C. If a multiple-error program contains errors which can each be repaired by a single edit operation (simple errors), then Recovery Method 1 will make that repair for those errors which are far enough apart, that is separated by $\rho$ symbols or more. (If $\rho$ symbols of input can be accepted after a single-operation repair, that repair is chosen.) But if two errors occur within $\rho$ symbols of each other, the method has to resort to secondary recovery, as no single operation will cause sufficient input to be accepted during primary recovery. Secondary recovery cannot find a single-operation repair as it employs a phrase-level technique. Hence if simple errors occur separated by between 5 and 10 input symbols, Recovery Method 1 will find the single-operation repairs when $\rho$ is 5 but not when $\rho$ is 10. Fig. 5.12 gives an example of a program from the Ripley collection in which there are two simple errors separated by seven symbols. Recovery Method 1 makes two edit operations when $\rho$ is 5 and four edit operations when $\rho$ is 10. The programs in the Ripley collection are quite short and errors in the multiple-error programs tend to occur in clusters. These clusters of errors explain the degradation of performance of Recovery Method 1 when lookahead is increased.

```
1  program p(input, output);
2  var key, record: array[1..limit] if akfa;
3  begin x:=1 end.
```

Recovery Method 1 ($\rho = 5$):

```
Line 2: syntax error
var key, record: array[1..limit] if akfa;
.........^
'record' replaced by 'YID'
Line 2: syntax error
var key, YID: array[1..limit] if akfa;
.............................^
'if' replaced by 'of'
```

Recovery Method 1 ( $\rho = 10$):

```
Line 2: syntax error
var key, record: array[1..limit] if akfa;
.........^
'key, record' replaced by 'id_list'
Line 2: syntax error
var id_list: array[1..limit] if akfa;
.............................^
'if' replaced by 'of'
```

**Fig. 5.12.** Cluster of errors in a Pascal program.

When the length $\sigma$ of repair which Recovery Method 2 generates is increased from 4 to 5, performance is degraded slightly: from excellent recovery on 69% of Pascal (weighted)

programs down to 64%, and from 67% down to 65% for C programs. This degradation is not significant. But the degradation which occurs when $\sigma$ is increased to 7 is significant, down to excellent recovery on 48% of Pascal programs (although the unweighted set shows excellent recovery on 57% for $\sigma = 4$ down only slightly to 52% for $\sigma = 7$). Figure 5.9 clearly shows that most of the performance decline is due to poor performance on single-error, heavily weighted programs, from excellent recovery on 77% of the Pascal single-error (weighted) set down to 48%. Inspection of the raw data shows that the decline is mainly due to different repairs made to just one program, shown in Fig. 5.13. This program has the largest weight of the collection at 72 (the second largest weight is 18): it is not a surprise that the error is a missing semi-colon! What is surprising is the repair made by the method with $\sigma$ of 7. This is another example of a repair that the author cannot explain - in theory the single-operation repair of the insertion of the missing semi-colon should have been chosen.

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

Recovery Method 2 ($\sigma = 4$):

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

Recovery Method 2 ($\sigma = 7$):

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

**Fig. 5.13.** Pascal program with largest weight.

## 5.4 COMPARISON WITH OTHER METHODS

### 5.4.1 Methods for Performance Evaluation

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" (Sippu and Soisalon-Soininen, 1983) or "most plausible repair" (Stirling, 1985). 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 models of minimum-distance and prefix-defined 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 our 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) 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 (60% of the Ripley suite) 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, the fragment

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

might be corrected by a programmer to

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

using knowledge of both syntax rules (no use of expressions in `const` declarations) and

also simple arithmetic. (It is also possible that questions would be asked about the necessity for declaring a second constant.) This error is described by Ripley and Druseikis as an "extra + 1" but a *competent* programmer would be unlikely to correct it simply by deleting those extra symbols. The error in the statement

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

is described 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 programs in the Ripley suite have been shortened and altered so that they are not in themselves plausible programs, so it is not possible to find a plausible repair. Stirling 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.

### 5.4.2 Results of Performance Evaluation

In order to compare our results of performance evaluation with those of other methods, we return to Fig. 2.1, Chapter 2, Section 2.6, which shows the percentage of recovery actions taken on the (unweighted) Ripley suite which are *acceptable*. The definition of acceptable given there approximates closely with our measure of the percentage of programs for which the number of edit operations equals the number of minimum-distance errors plus or minus one, shown in Fig. 5.9. The relevant figures from Fig. 5.9 are combined with Fig. 2.1 to give Fig. 5.14.

The majority of methods give acceptable performance for 70 to 80% of Pascal programs. It should be noted that our methods also give acceptable performance in this range for C programs (e.g. 75% for Recovery Method 1, $\rho = 5$). Recovery Method 1 achieves a lower

standard than all other methods for Pascal but Recovery Method 2 achieves better than average performance.

| | |
|---|---|
| Recovery Method 1, $\rho = 10$ | 51% |
| Recovery Method 1, $\rho = 5$ | 57% |
| Stirling (a) | 66% |
| Stirling (b) | 66% |
| Sippu and Soisalon-Soininen | 67% |
| Pennello and DeRemer | 70% |
| Wirth | 72% |
| Recovery Method 2, $\sigma = 4$ | 74% |
| Recovery Method 2, $\sigma = 7$ | 77% |
| IBM Pascal/VS | 77% |
| Pai and Kieburtz | 77% |
| Recovery Method 2, $\sigma = 6$ | 78% |
| Recovery Method 2, $\sigma = 5$ | 79% |
| Anderson and Backhouse | 79% |
| Spenke et al | 91% |
| Burke and Fisher | 98% |

**Fig. 5.14.** Percentage of recovery actions which are acceptable.

We can also compare methods for excellent performance, where the classification excellent includes the categories *good* of Anderson et al and Stirling, *excellent* of all other authors ("same as a competent programmer might make"), and for our methods, the percentage of programs for which the number of edit operations equals the number of minimum-distance errors (shown in Fig. 5.8). Fig. 5.15 shows the percentage of recovery actions taken (percentage of programs) for the Ripley suite which are classified as excellent.

The best performance of our recovery methods according to this measure (Recovery Method 1, $\rho = 5$) is better than six of the other authors' schemes and worse than four. It should also be noted that this method gives excellent performance for 73% of the collection of C programs.

| | |
|---|---|
| Sippu and Soisalon-Soininen | 36% |
| Stirling (a) | 38% |
| Stirling (b) | 40% |
| Pennello and DeRemer | 42% |
| Wirth | 45% |
| Recovery Method 1, $\rho = 10$ | 50% |
| Recovery Method 2, $\sigma = 7$ | 52% |
| Pai and Kieburtz | 52% |
| Recovery Method 2, $\sigma = 5$ | 55% |
| Recovery Method 2, $\sigma = 6$ | 55% |
| Recovery Method 2, $\sigma = 4$ | 57% |
| Recovery Method 1, $\rho = 5$ | 57% |
| Anderson and Backhouse | 57% |
| Boullier and Jourdan | 75% |
| Spenke et al | 77% |
| Burke and Fisher | 78% |

**Fig. 5.15.** Percentage of recovery actions which are excellent.

# Chapter 6 Conclusions

## 6.1 SUMMARY

In the preceding chapters the need was argued for good syntax error handling schemes and their automatic incorporation into practical parser-generators. A model for syntax errors based on prefix-defined errors was constructed. It was shown that common programming languages allow an infinite number of programs that differ in a single symbol only and that it is undecidable whether a CFL allows a better than arbitrary choice of edit operations at a point of error. For Pascal, minimum distance errors and prefix-defined errors do not necessarily coincide, and practical $O(n)$ parsers cannot detect errors as well as $O(n^3)$ parsers. Sets of equivalent insertions for common programming languages were exhibited. Two new practical methods for error recovery were presented, using different approaches to the problem. The first method used a two-stage method of local repair followed by phrase-level recovery; the second used a single-stage method that aimed to provide a practical minimum-distance repair, i.e. locally minimum-distance. Criteria for assessing error handling schemes were established and the two new methods were assessed using those criteria. A new method for performance evaluation was presented and used to evaluate performance of the two new schemes, and to compare their performance with other authors' schemes.

## 6.2 CONCLUSIONS

The model for syntax errors based on the concept of prefix-defined correction has proved useful as a theoretical yardstick. It has turned out in practice to be similar to the minimum-distance model for the programming languages investigated. The model is intended to measure the shortest way a practical (correct prefix) parser could transform an incorrect string into a sentence. Other authors have attempted to construct models for errors based on concepts other than minimum distance (see for example Wetherell (1977) and Sippu

and Soisalon-Soininen (1980)) but these have not been widely used. The model presented here is unlikely to prove any more useful until further insight into the nature of errors is gained.

The theoretical results established in Chapter 3 add to the existing body of knowledge about syntax errors. In part, they confirm the difficulties facing the designer of a syntax error recovery scheme and the lack of methods for deciding between parsing method; language and grammar. Despite these pessimistic results, we have argued that it should be possible to build in good error handling automatically to parsers. A parser-generator has been enhanced with two new error recovery schemes and used to build compilers for many languages. The implementation of the first recovery scheme has been favourably received by several compiler-writers. The use of minimum-distance correction in a practical tool is original. The two new methods for error recovery compare well with previous methods in terms of performance, and are superior in terms of their practical application. The generation of clear, informative error messages has been completely automated. The resulting tools should be of interest and practical use to compiler writers.

## 6.3 SUGGESTIONS FOR FURTHER WORK

### 6.3.1 Alterations and Extensions

The first group of suggestions for further work is directly related to the work presented in this thesis and comprises alterations and extensions. Firstly, it is unsatisfactory that the implementation of Recovery Method 2 has produced some unexpected results (repairs of distance two in some cases where a single repair is expected). The initial step in an investigation of the software would be to augment the code to produce diagnostic information on the repairs constructed and their distances from the input, and the sequence of states entered by the parser during error recovery. Diagnostic information about states including legal symbols, shift states and reduce actions is already available. This information would

then be compared with the expected sequences of states and generated repairs for the programs giving unexpected results, with the aim of identifying where the implementation diverges from the design.

Secondly, the Ripley collection of student Pascal programs may be getting out of date, and our collection of C programs is small. In order to build an up-to-date database of inputs for error handling schemes, more programs could be collected and analysed. It would be desirable to make collections for several common programming languages, including Pascal, C, Modula-2, Ada and occam, building the collections up over different periods and in different working environments so that representative sets were obtained.

Thirdly, automation of the evaluation process is suggested. This involves an implementation of the Aho and Peterson algorithm for minimum-distance error correction, and an extension of the implementations of Recovery Methods 1 and 2 so that a count of the number of edit operations made for the chosen repair is output. These software tools could then be used together to produce automatically the information which is processed to obtain performance figures such as those presented in Chapter 5. Work on an algorithm for producing a minimum prefix-defined errror correction, given a CFG and a string as input, is suggested. An implementation of the algorithm could then be used in the automated evaluation process. A possible starting point for the work is the Cocke-Younger-Kasami algorithm (Hopcroft and Ullman, 1979, p. 139) for determining membership. It appears probable that the dynamic programming technique of this algorithm can be extended to simulate a series of parallel parses on altered input strings.

### 6.3.2 Future Directions

The second group of suggestions for further work is broader in scope than the first group and indicates possible future directions for work on error handling. When this work was started, we perceived a genuine need for good syntax error handling in a programming

environment which included tools such as general-purpose text editors, non-interactive compilers and interpreters. Since that time, advances in hardware technology have given rise to a new generation of machines incorporating RISC chips, parallel architectures, high bandwidth communication and cheap memory. These advances are paralleled by theoretical and practical developments in software, giving rise to a new generation of programming environments. The paradigm of object-oriented design and programming has led to powerful new languages such as C++ and Smalltalk. IPSEs and CASE tools incorporate use of formal methods and structured methods in interactive graphics interfaces. More power has been placed on the programmer's desk, with more sophisticated software tools to exploit that power. In the programming environments of the future, error handling will be as important as it is today, and work is needed on the design of appropriate error handling.

Aho, Sethi and Ullman (1986, p. 164) write that "with the increasing emphasis on interactive computing and good programming environments, the trend seems to be toward simple error-recovery mechanisms"; but we argue that there is still a need for sophisticated error-recovery mechanisms within interactive systems. Interactive programming still requires parsing and translation of the user's input; one of the main differences between interactive programming systems and batch compilers is that the unit of interaction between user and computer is smaller, typically an expression or function rather than a program unit. Fig. 6.1 gives several examples of use of an existing interactive environment for a functional programming language, Standard ML (Harper, Milner and Tofte, 1988). In the figure the user's input is shown in italics. There are two system prompts, a primary prompt '-' and a secondary prompt '=' to indicate that further input is expected. The error messages output by the system do not meet accepted criteria: they are not clear or informative, they do not succeed in directing the user's attention to the point of error, and they are not oriented towards the user but are expressed in terms of the internal parsing mechanisms.

```
- +1;
Parse error: Was expecting ";"
In: ... < ? > +
- 5+;
Parse error: Was expecting an Expression
In: ... 5 + < ? > ;
- fun fact (x)
=     if x < 1 then 1
=     else (fact(x-1) * x;
= ;
Parse error: Insufficient repetition
In: ... 1 then 1 else ( fact ( x - 1 ) * x ; < ? > ;
```

**Fig. 6.1.** An interactive session with Standard ML.

Another example where further work on error handling can be applied is the integrated programming environment. Program development usually consists of a cycle of editing, compiling, executing and debugging, traditionally achieved with separate utilities such as a general-purpose text editor, a compiler, a loader and a debugger, each with their own user interface. Integrated programming environments are intended to support program development through a unified user interface and typically include a structure or context-sensitive (language-specific) editor. This model of program development uses the paradigm of immediate computation; the editor analyses and translates the source code as it is entered, handling syntactic and static semantic errors. Whatever technique for parsing is used, there is a requirement for good interactive error handling which can be invoked by the editor.

New tools such as the Synthesizer Generator of Reps and Teitelbaum (1988) are being developed in order to assist in the design and production of new programming environments.

These tools are typically used to build syntax and semantics analysers and attribute evaluators from language specifications based on the concept of an attribute grammar (Knuth, 1968). The Synthesizer Generator itself uses *yacc* to build the parser module of the editors it constructs. The resulting syntax error handling is simple: when an error is detected by the parser, the message `"Syntax error"` is displayed and the cursor is located at the last character of the incorrect token, i.e. the prefix-defined error. No indication of the nature of the error is given and no attempt at recovery or repair is made. More sophisticated error handling should provide better recovery and better error messages. We believe that our methods form a basis for good error handling schemes, but further work is needed in order to solve the problem of how to choose an appropriate repair when there is little upcoming input to be used in making that choice. Work is also needed on a separate but related problem, the design of an appropriate user interface in the interactive handling of errors. The unit of interaction is a parameter to both these problems. The first problem may be adequately solved by use of Recovery Method 2. Generation of continuation strings and selection of the one closest to the actual input, however short the input may be, is a technique worth evaluating in the interactive environment.

An application for interactive parsers with error recovery is in the use of formal methods for the design and implementation of real-time systems. There is current research on the transformation of functional specifications of real-time systems into concrete implementations (Moitra and Joseph, 1989). Such transformations require the static analysis of specifications and implementations, to ensure for example that constraints are met by the transformed program. Tools to perform static analyses will be needed, and we suggest the design of interactive tools, using meta-tools such as the Synthesizer Generator mentioned above. A study of the role and methods of error handling, in the context of interactive analysis of formal specifications, would form a part of this work.

# Appendix A Extract from "Minimum Distance Error Correction"

Extract from "Minimum Distance Error Correction" (Dain, 1987), pp. 3-7.

## Overview of the Method

Each state of an LR parser contains information on all the possible moves from that state: which input symbols can be shifted and which reductions are possible (LR(1) parsers inspect one symbol of lookahead to determine wheter a reduction is legal). The parser detects an error in the input when there is no legal move (neither shift nor reduce) on the current configuration, and invokes the recovery method. The method generates *legal continuation strings* for the input already parsed, i.e. prefixes of legal suffixes of the parsed input; the actual remaining input is not inspected during generation of the continuation strings. One of the continuation strings is then chosen as the repair to the actual input; the minimum distance measure from the actual input is used to choose the best of the continuation strings.

The continuation strings are formed from successive legal shift moves of the parser. Each possible shift move will cause an input symbol to be concatenated to a continuation string. The method constructs *recovery configurations* of the parser which consist of a parser stack and a continuation string. The initial recovery configuration is formed from the parser stack at the point of detection of error and the empty string. For each recovery configuration, all legal moves from the top state of the stack are considered. A shift move gives rise to a new recovery configuration consisting of the stack with the shift state pushed on and the previous configuration's continuation string with the shift symbol concatenated. A reduce move gives rise to a new recovery configuration consisting of the reduced stack and the previous continuation string (there is no inspection of a look-ahead symbol).

# The LR Parsing Machine with Minimum Distance Error Recovery

A context-free grammar $G$ is represented by a tuple $(N, \Sigma, P, S)$ where

$N$ is a finite alphabet of non-terminals

$\Sigma$ is a finite alphabet of terminals

$P$ is a finite set of productions $P : N \rightarrow (N \cup \Sigma)^*$

$S$ in $N$ is the start symbol.

An LR(0) parsing automaton $M$ for an LR(0) grammar $G$ may be represented by a tuple

$(Q, \Sigma, q_0, \delta, \gamma)$ where

$Q$ is a finite set of states (the stack alphabet)

$q_0$ in $Q$ is the start state

$\Sigma$ is the finite input alphabet (terminals of $G$)

$\delta$ is the ACTION transition function

$$\delta : Q \times \{\Sigma \cup \lambda\} \rightarrow SHIFT \times Q \cup REDUCE \times P \cup ACCEPT \cup ERROR$$

$\gamma$ is the GOTO transition function $\gamma : Q \rightarrow Q$

Moves of the parsing automaton are either shift moves which consume one input symbol and push a state onto the stack, or reduce moves which consume no input and replace zero or more of the top states of the stack with a new state, or halting moves which accept or announce error. A configuration of the parser stack and input is represented by an instantaneous description (ID) of the form $[\, q_0 \ldots q_m, a_j \ldots a_{j+k} \,]$ where the $q_i$ represent the stack ( $q_m$ the top state) and the $a_i$ represent the remaining input. The symbol |- denotes the relation "move in one step" on IDs. A shift move is then denoted by successive IDs

$$[\, q_0 \ldots q_m, a_j a_{j+1} \ldots a_{j+k} \,] \vdash [\, q_0 \ldots q_m q', a_{j+1} \ldots a_{j+k} \,]$$

where $\delta(q_m, a_j) = (SHIFT, q')$

A reduce move is denoted by successive IDs

$$[\, q_0 \ldots q_{m-n} \ldots q_m, a_j \ldots a_{j+k} \,] \vdash [\, q_0 \ldots q_{m-n} q', a_j \ldots a_{j+k} \,]$$

where $\delta(q_m, \lambda) = (REDUCE, A \rightarrow \alpha)$, $|\alpha| = n$, and $\gamma(q_{m-n}) = q'$

A recovery configuration of the parser is represented by an ID $[\, q_0 \ldots q_m, u \,]$, $u \in \Sigma^*$, where the $q_i$ represent the stack as before and $u$ represents the continuation string (we use [ and ] in place of [ and ] to distinguish IDs representing recovery configurations from IDs representing ordinary configurations). The relation "move in one step" on recovery configurations, denoted by the symbol $\vdash_R$, is defined analogously to $\vdash$ on parser configurations, as follows.

If $\delta(q_m, a) = (SHIFT, q')$, $a \in \Sigma$, then $[\, q_0 \ldots q_m, u \,] \vdash_R [\, q_0 \ldots q_m q', ua \,]$

If $\delta(q_m, \lambda) = (REDUCE, A \rightarrow \alpha)$, $|\alpha| = n$, and $\gamma(q_{m-n}) = q'$, then

$$[\, q_0 \ldots q_m, u \,] \vdash_R [\, q_0 \ldots q_{m-n} q', u \,]$$

Let $\vdash_R^*$ denote the reflexive and transitive closure of $\vdash_R$. Let the configuration of the parser when error is detected be denoted by ID $[\, q_0 \ldots q_e, a_j \ldots a_{j+k} \,]$. Then the set $\Theta_L$ of continuation strings of length $L$ is given by

$$\Theta_L = \{\, u \mid u \in \Sigma^L, [\, q_0 \ldots q_e, \lambda \,] \vdash_R^* [\, q_0 \ldots q_n, u \,] \,\}$$

## Generation of the Continuation Strings

We give a design in Pascal for the algorithm to generate the continuation strings. First, we outline the data types to be used.

TOKEN represents an input symbol (terminal or lexical token) of the grammar.

PRODUCTION represents a production of the grammar.

STRING abstracts a string of tokens with operations *print, concat, length.*

TOKENSET abstracts a set of tokens with iterator *nextT*.

PRODUCTIONSET abstracts a set of productions with iterator *nextP*.

STACK abstracts a stack of (LR(1)) parser states with operations

shift: STACK × TOKEN → STACK

reduce: STACK × PRODUCTION → STACK

LegalShifts: STACK → TOKENSET

LegalReductions: STACK → PRODUCTIONSET

GenRepairs: STACK × STRING → {}

The operations LegalShifts and LegalReductions inspect the top state of the stack to return the tokens which it is possible to shift and the productions by which it is possible to reduce respectively.

The algorithm is implemented by the recursive procedure *GenRepairs*. The input to the algorithm is the current configuration of the parser, consisting of the current stack, passed as a parameter, and the actual remaining input to the parser. The output is a sequence of strings of tokens (the continuation strings).

```
1   procedure GenRepairs(stack: STACK, continuation: STRING);
2   const REPAIRLENGTH = 10;  {desired length of continuation }
3   var shifts: TOKENSET;  { legal shifts from current state }
4         t: TOKEN;  { next symbol for shift move }
5         reductions: PRODUCTIONSET;  { legal reductions from current state }
6         p: PRODUCTION;  { next production for reduce move }
7   begin
8         if length(continuation) >= REPAIRLENGTH
9               then print(continuation)
10              else  begin
11                    { compute legal shifts }
12                    shifts := LegalShifts(stack);
13                    t := nextT(shifts);
14                    while t <> 0 do
15                          begin
16                          GenRepairs(shift(stack, t), concat(continuation, t));
17                          t := nextT(shifts)
18                    end;
19              { compute legal reductions }
20              reductions := LegalReductions(stack);
21              p := NextP(reductions);
22              while p <> 0 do
23                    begin
24                          GenRepairs(reduce(stack, p), continuation);
25                          p := nextP(reductions)
26                    end
27        end
28  end;
```

## Computing the minimum distance

We use the Fischer-Wagner algorithm [13] to compute the edit distance of two strings $v$ of lengths $m$ and $n$ over a finite alphabet, with error transformations insert, delete and replace. This algorithm constructs an $m \times n$ matrix $M$ where $M(i, j)$ gives the minimum cost of transforming the prefix of length $i$ of string $u$ into the prefix of length $j$ of string $v$. The algorithm has time complexity $O(mn)$ (Masek and Paterson [9]) improve this algorithm to one of $O(n \,.\, max(1, m/log\, n))$ by use of the Four Russians' idea).

## Restarting the parser

In order to continue parsing the remaining input after having formulated a good repair, it is desirable for the parser to be synchronized with the input. We wish to identify the most appropriate continuation string with which to replace a portion of the input. If continuation strings of length $L$ are generated, then the minimum distances of the continuation strings from a prefix of length $2L$ of the upcoming input is computed. The continuation string with the smallest entry in the last row of the Fischer-Wagner matrices is found; if the column number of this entry is $c$ then the next $c$ tokens on the input are replaced with the chosen continuation string. We choose the samllest entry in the last row, over all the Fischer-Wagner matrices, rather than the smallest entry in the last column of the last row, because it is possible to have a continuation string whose last Fischer-Wagner entry $M(L, 2L)$ is greater than that of another, less suitable, continuation. There may be more than one continuation string with the same smallest entry $x$ in the last row and in this case we choose the continuation which will replace most of the upcoming input, i.e. the entry with the largest column number $c$. Consider for example the Fischer-Wagner matrices for actual input *abcdef*, $L = 3$, repairs *gcd*, *gab* :

| | $\lambda$ | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ |
|---|---|---|---|---|---|---|---|
| $\lambda$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| $g$ | 1 | 1 | 2 | 3 | 4 | 5 | 6 |
| $c$ | 2 | 2 | 2 | 2 | 3 | 4 | 5 |
| $d$ | 3 | 3 | 3 | 2 | 2 | 3 | 4 |

| | $\lambda$ | $1$ | $2$ | $3$ | $4$ | $5$ | $6$ |
|---|---|---|---|---|---|---|---|
| $\lambda$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| $g$ | 1 | 1 | 2 | 3 | 4 | 5 | 6 |
| $a$ | 2 | 1 | 2 | 3 | 4 | 5 | 6 |
| $b$ | 3 | 2 | 1 | 2 | 3 | 4 | 5 |

Here we choose the repair *gab* in preference to *gcd*, because the former allows the possibility of completing the repaired input with the string commencing *cdef*, giving an overall repair of a single insertion, whereas the latter forces at least a deletion and a replacement.

If we denote the Fischer-Wagner matrix of a continuation string $u$ by $M_u$, then the required repair $w$ in $\Theta_L$ satisfies

If $M_w(L, c) = \min \{ M_w(L, i) \mid i = 1, ..., 2L \}$

then for all $u$ in $\Theta_L$; $M_u(L, i) \geq M_w(L, c)$ for $1 \leq i \leq 2L$,

and if $M_u(L, i) = M_w(L, c)$ then $i \leq c$.

If an error configuration of the parser $[ q_0 \ldots q_e, a_j \ldots a_{j+k} ]$ gives rise to the choice of repair $w$,

$$w \in \Theta_L, [ q_0 \ldots q_e, \lambda ] \vdash_R^* [ q_0 \ldots q_n, w ],$$

then the parser is restarted in configuration $[ q_0 \ldots q_n, a_{j+c} \ldots a_{j+k} ]$ where $c$ satisfies

$$M_w(L, c) = \min \{ M_w(L, i) \mid i = 1, ..., 2L \}$$

## Pruning the Search Tree

An upper bound on the number of possible continuation strings is $|\Sigma|^L$ where $L$ is the length of repair desired. The configurations generated by the method can be viewed as a tree with the state in which error is detected and the empty continuation string at the root. We wish to reduce the size of this search tree. The first idea for pruning the tree is to use a depth-first search of the tree and to record a current best repair against which other potential repairs are measured and possibly discarded early. One configuration is followed to completion and its minimum distance from the actual input is recorded. This is the current best so far. Generation of the next configuration is started, simultaneously constructing the continuation string's Fischer-Wagner matrix. If at any stage all the entries in the current row of the matrix are greater than or equal to the current best, then this branch of the tree can be pruned. If the generation continues to completion, and its minimum distance is less than the current best, then this repair becomes the new best.

The second idea is to refine this method by estimating a likely best for the first continuation string. One way of estimating such a string is to mark the current position in the input, choose a shift or reduce on a symbol which appears, nearest, in the input, mark this new position in the input and repeat. This estimation simulates deletions; a simulation of insertions and replacements is needed. The third idea simulates these operations in a breadth-first search of the tree. At the first level of the tree, all possible configurations are generated. At the second level, we choose only those configurations whose accessing symbol (shift or reduce) appears nearest in the input; this simulates an insertion followed by zero or more deletions. This gives the following heuristic: (i) always choose the accessing symbol which appears next in the input if possible; (ii) if there is none such, simlate a single insertion followed by zero or more deletions as above. A multiple insertion can be simulated by increasing the number of levels of the search tree at which all configurations are generated from one level to $k$ levels, for an insertion of length $k$ symbols.

## Appendix B Parser Template

```
// template parser for yacc
// C++, 4.1 BSD
// (c) J A Dain, University of Warwick, 1986
// modified JAD Jan 1987

#include <stream.h>
//yacc declaration
#define YYFLAG -1000

typedef short YACCTABLE;          // yacc array type
typedef short STATE;              // parser state
typedef int PRODN;                // production
typedef int TOKEN;                // input token

//-------------------------------------------------------------
// obs and stacks
// M F Rafter, UoW Jan 1987
// modified for stacks  J A Dain, UoW Jan 1987

#define NO_OBJ((ob *)0)      /* non-existent object */

class ob                          // linked list of objects
{
   ob *nxt;
public:
   void add(ob *obj) { if (this != NO_OBJ) nxt = obj; }
   ob *next() { return ((this == NO_OBJ) ? NO_OBJ : nxt); }
};

class stack_ob          // a stack of obs
{
   ob* top;             // top of stack, starts as NO_OBJ
   ob* bottom;          // bottom of stack, starts as NO_OBJ
public:
   stack_ob() { top = bottom = NO_OBJ; }
   void push(ob* obp); // push obp onto stack
   ob* pop();           // pop top ob from stack; return NO_OBJ
```

```
                        // if empty
    int isempty() { return(top == NO_OBJ); } // true if stack is
                                             // empty
    ob* peek() { return top; } // peek at top ob on stack;
                               // NO_OBJ if empty
};

void stack_ob::push(ob *obp) // push *obp onto top of stack_ob
{
    if (obp == NO_OBJ) return;
    obp->add(top);
    top = obp;
    if (bottom == NO_OBJ)
        bottom = obp;
}

ob* stack_ob::pop()          // pop object from top of stack_ob
                             // return NO_OBJ if stack_ob empty
{
    ob* obp = top;
    if (obp != NO_OBJ) {
        top = obp->next();
        if (top == NO_OBJ)
            bottom = NO_OBJ;
        obp->add(NO_OBJ);
    }  return(obp);
}

//-----------------------------------------------------------
// stack of parser states for yaccpar

#define NO_STATE ((state_ob *) NO_OBJ)

class state_ob : ob {        // parser state
    STATE statename;
public:
    state_ob(STATE s) { statename = s; }
    void print() { cerr << statename << "\n"; }
    STATE peek() { return statename; }
```

```
} ;

class state_stack : public stack_ob {    // stack of state_obs
public:
   void push(state_ob *p) { stack_ob::push((ob *)p); }
         // pop and peek return NO_STATE if stack is empty:
   state_ob *pop() { return((state_ob *)(stack_ob::pop())); }
   state_ob *peek() { return((state_ob *)(stack_ob::peek())); }
} ;

class parser_state_stack : public state_stack {
   // stack of parser states
public:
   void push(STATE s);
   STATE pop();
   STATE peek();
} ;

void parser_state_stack::push(STATE s)
{
   state_ob *p = new state_ob(s);
   state_stack::push(p);
 }

STATE parser_state_stack::pop()
{
   state_ob *p = state_stack::pop();
   if (p == NO_STATE) return 0;
   short s = p->peek();
   delete(p);
   return(s);
   }

   STATE parser_state_stack::peek()
   {
      state_ob *p = state_stack::peek();
      return (p == NO_STATE) ? 0 : p->peek();
   }
   //-----------------------------------------------------------
```

```
enum parse_action { ACC,ERR,SHIFT,REDUCE } // parser actions
struct action_entry {          // entry in parser action table -
   parse_action t_action;      // parser action
   STATE t_state;              // shift to this state
   PRODN t_production;         // reduce by this production
} ;
typedef YACCTABLE goto_entry;  // entry in parser goto table

extern action_entry LRaction(STATE,TOKEN);//action table entry
extern action_entry LRdefault(STATE, TOKEN);
   // default action table entry
extern goto_entry LRgoto(STATE, PRODN); // goto table entry
extern int LRprodn_len(PRODN);     // length of production RHS

const int MAXDEPTH = 200;    // size of parser stacks
const TOKEN NOTOKEN = -5;    // denotes input token not set

class parser {
   parser_state_stack pss;          // stack of states
// Here follows an example of poor style: the semantic value
// stack is NOT realized with an ADT, but implemented
// explicitly with an array of YYSTYPEs.  YYSTYPE is a
// user-defined type (in the input to yacc, %union - defaults
// to integer) and the semantic action code makes explicit use
// of the array yyv.
   YYSTYPE yyv[MAXDEPTH];    // stack of semantic values
   YYSTYPE *vp;              // current top of value stack
   TOKEN tok;                // current input token
   void shift(STATE state);  // shift state onto stack
   void reduce(PRODN p);     // reduce stack by production p
   void print();             // print top of state stack
public:
   parser(STATE i = 0);      // initial state default is 0
   parse_action move();      // make a move of the parser
} ;

parser::parser(STATE i)
{
   pss.push(i);          // initialize state stack with state i
```

```
    vp = &yyv[0];
    tok = NOTOKEN;
}

void parser::shift(STATE state)          // shift to state
{
    pss.push(state);
    if (vp >= &yyv[MAXDEPTH-1]) {
        cerr << "Parser value stack overflow\n";
        return;
    }
    *++vp = yyval;
}

void parser::reduce(PRODN p)             // reduce by production p
{
    YYSTYPE *yypvt = vp;
    int p_len = LRprodn_len(p);
        // pop stacks
    for (int i = 0; i < p_len; i++)
        (void) pss.pop();
vp -= p_len;
yyval = vp[1];
PRODN yym = p;                  // for semantic actions
    // find goto state:
STATE gotostate = LRgoto(pss.peek(), p);
    // here,yacc embeds the user's code for semantic actions:
switch (yym) {
    $A
}
shift(gotostate);
}

void parser::print()
{
    cerr << "parser top of stack: " << pss.peek() << "\n";
}

parse_action parser::move()
```

```
{
   if (tok == NOTOKEN) tok = yylex();
   action_entry e = LRaction(pss.peek(), tok);
   switch (e.t_action) {
      case ACC:
         return ACC;
      case ERR:
         return ERR;
      case SHIFT:
         yyval = yylval;
         shift(e.t_state);
         tok = NOTOKEN;       // consume token from input
         return SHIFT;
      case REDUCE:
         reduce(e.t_production);
         return REDUCE;
   }
}

//-----------------------------------------------------------
// LR table functions -
// look up entries in the LR action,goto and production tables
// LRaction: STATE X TOKEN -> action_entry
// LRdefault:STATE X TOKEN -> action_entry
// LRgoto:   STATE X PRODN -> goto_entry
// LRprodn_len: PRODN -> int

action_entry LRaction(STATE state, TOKEN ch)
{
   action_entry t;
   t.t_action = ACC;
t.t_state = 0;
t.t_production  = 1;
YACCTABLE n = yypact[state];
if (n <= YYFLAG)
   return LRdefault(state, ch);
if (ch > 0)
   n += ch;
if (n < 0 || n >= YYLAST)
```

```
        return LRdefault(state, ch);
    n = yyact[n];
    if (yychk[n] == ch) {
        t.t_action = SHIFT;
        t.t_state = n;
        return t;
    }
    else return LRdefault(state, ch);
}

action_entry LRdefault(STATE state, TOKEN ch)
{
    const YACCTABLE EXCEPTIONS = -2;      // yacc magic number
    action_entry t;
    t.t_action = ACC;
    t.t_state = 0;
    t.t_production = 1;
    YACCTABLE n = yydef[state];
    if (n == EXCEPTIONS) {
        if (ch < 0) ch = 0;
        // look through exception table
        YACCTABLE *x;
        for (x = yyexca; (*x != -1) || (x[1] != state); x+= 2)
            ;
        while (*(x += 2) >= 0) {
            if (*x == ch) break;
        }
        if ((n = x[1]) < 0)              // ACC
            return t;
    }
    if (n == 0) {
        t.t_action = ERR;
        return t;
    }
    else {
        t.t_action = REDUCE;
        t.t_production = n;
        return t;
    }
```

```
}

goto_entry LRgoto(STATE state, PRODN p)
{
   YACCTABLE n = yyr1[p];
   YACCTABLE j = yypgo[n] + state + 1;
   if (j >= YYLAST || yychk[yyact[j]] != -n)
      return yyact[yypgo[n]] ;
   else return yyact[j] ;
   }

   int LRprodn_len(PRODN p)
   {
      return yyr2[p];
   }

   //-----------------------------------------------------------

   void yyparse()
   {
   parser P;
   while (1) {
      parse_action a = P.move();
      if (a == ACC) {
         break;
      }
      else if (a == ERR) {
         yyerror ( "syntax error\n");
         break;
      }
   }
   }
```

# Appendix C Results of Performance Evaluation

| No. of edit operations performed by recovery method | No. 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 |

| No. of edit operations performed by recovery method | No. of prefix-defined 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 | 1 | 3 | 1 | | | | | | |
| 9 | | | | | | | 1 | | |
| 10 | | | | | | | | | |
| 11 | | 1 | | | | | | | |
| 12 | | 1 | | | | | | | |
| >12 | 7 | 5 | 9 | 2 | 1 | 3 | 4 | | |
| Total | 68 | 24 | 15 | 2 | 2 | 3 | 5 | 0 | 0 |

**Fig. C.1.** Numbers of programs, edit operations against errors:

Pascal, Recovery Method 1, $\rho = 5$.

| No. of edit operations performed by recovery method | No. of minimum-distance errors | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 58 | | | | | | | | |
| 2 | | 3 | | | | | | | |
| 3 | 1 | 1 | | | | | | | |
| 4 | 1 | 2 | | | | | | | |
| 5 | 1 | 1 | 2 | | | | | | |
| 6 | | 2 | | | | | | | |
| 7 | | 1 | 1 | | | | | | |
| 8 | 2 | 2 | 1 | | | | | | |
| 9 | | 1 | | | | | | | |
| 10 | | | | | | | | | |
| 11 | | 1 | 1 | | | | | | |
| 12 | | 2 | | | | | | | |
| >12 | 9 | 8 | 8 | 3 | 2 | 3 | 2 | | |
| Total | 72 | 24 | 13 | 3 | 2 | 3 | 2 | 0 | 0 |

| No. of edit operations performed by recovery method | No. of prefix-defined errors | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 58 | | | | | | | | |
| 2 | | 3 | | | | | | | |
| 3 | 1 | 1 | | | | | | | |
| 4 | 1 | 2 | | | | | | | |
| 5 | 1 | 1 | 2 | | | | | | |
| 6 | | 2 | | | | | | | |
| 7 | | 1 | 1 | | | | | | |
| 8 | | 3 | 2 | | | | | | |
| 9 | | 1 | | | | | | | |
| 10 | | | | | | | | | |
| 11 | | 1 | 1 | | | | | | |
| 12 | | 2 | | | | | | | |
| >12 | 7 | 7 | 9 | 2 | 2 | 3 | 4 | | |
| Total | 68 | 24 | 15 | 2 | 2 | 3 | 5 | 0 | 0 |

**Fig. C.2.** Numbers of programs, edit operations against errors: Pascal, Recovery Method 1, $\rho = 10$.

| No. of edit operations performed by recovery method | No. of minimum-distance errors | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 47 | | | | | | | | |
| 2 | 9 | 10 | 1 | | | | | | |
| 3 | 6 | 6 | 7 | | | | | | |
| 4 | 4 | 5 | 2 | 2 | | | | | |
| 5 | 2 | 1 | 1 | | 1 | | | | |
| 6 | 2 | | 2 | | 1 | 1 | 1 | | |
| 7 | | 1 | | 1 | | 1 | 1 | | |
| 8 | 1 | 1 | | | | | | | |
| 9 | | | | | | | | | |
| 10 | | | | | | | | | |
| 11 | | | | | | 1 | | | |
| 12 | 1 | | | | | | | | |
| >12 | | | | | | | | | |
| Total | 72 | 24 | 13 | 3 | 2 | 3 | 2 | 0 | 0 |

| No. of edit operations performed by recovery method | No. of prefix-defined errors | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 47 | | | | | | | | |
| 2 | 8 | 11 | 1 | | | | | | |
| 3 | 6 | 4 | 9 | | | | | | |
| 4 | 3 | 5 | 3 | 2 | | | | | |
| 5 | 1 | 1 | 1 | | 1 | | 1 | | |
| 6 | 2 | | 1 | | 1 | 1 | 2 | | |
| 7 | | 1 | | | | 1 | 2 | | |
| 8 | | | | | | | | | |
| 9 | | | | | | | | | |
| 10 | | 2 | | | | | | | |
| 11 | | | | | | | 1 | | |
| 12 | 1 | | | | | | | | |
| >12 | | | | | | | | | |
| Total | 68 | 24 | 15 | 2 | 2 | 3 | 5 | 0 | 0 |

**Fig. C.3.** Numbers of programs, edit operations against errors: Pascal, Recovery Method 2, $\sigma = 4$, $\tau = 8$.

| No. of edit operations performed by recovery method | No. of minimum-distance errors | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 48 | | | | | | | | |
| 2 | 14 | 10 | 1 | | | | | | |
| 3 | 3 | 8 | 6 | | | | | | |
| 4 | 1 | 3 | 1 | 2 | | | | | |
| 5 | 3 | 1 | 2 | | 1 | | | | |
| 6 | | | 2 | | 1 | | 1 | | |
| 7 | 1 | | | | | 2 | | | |
| 8 | | | 1 | | | | | | |
| 9 | 1 | 1 | | | | | | | |
| 10 | 1 | | | | | | | | |
| 11 | | | | | | | | | |
| 12 | | | | 1 | | 1 | | | |
| >12 | | 1 | | | | | 1 | | |
| Total | 72 | 24 | 13 | 3 | 2 | 3 | 2 | 0 | 0 |

| No. of edit operations performed by recovery method | No. of prefix-defined errors | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 48 | | | | | | | | |
| 2 | 12 | 11 | 2 | | | | | | |
| 3 | 3 | 7 | 7 | | | | | | |
| 4 | 1 | 3 | 1 | 2 | | | | | |
| 5 | 2 | 1 | 3 | | 1 | | | | |
| 6 | | | 1 | | 1 | | 2 | | |
| 7 | 1 | | | | | 2 | | | |
| 8 | | | 1 | | | | | | |
| 9 | | 1 | | | | | 1 | | |
| 10 | 1 | | | | | | | | |
| 11 | | | | | | | | | |
| 12 | | | | | | 1 | 1 | | |
| >12 | | 1 | | | | | 1 | | |
| Total | 68 | 24 | 15 | 2 | 2 | 3 | 5 | 0 | 0 |

**Fig. C.4.** Numbers of programs, edit operations against errors:

Pascal, Recovery Method 2, $\sigma = 5$, $\tau = 10$.

| No. of edit operations performed by recovery method | No. of minimum-distance errors | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 47 | 1 | 1 | | | | | | |
| 2 | 14 | 11 | 1 | | 1 | | | | |
| 3 | 3 | 5 | 6 | | | 1 | | | |
| 4 | 2 | 3 | 2 | 1 | | | | | |
| 5 | 3 | 2 | 2 | 1 | 1 | | | | |
| 6 | | | 1 | | | | 2 | | |
| 7 | 1 | | | | | 2 | | | |
| 8 | | 1 | | 1 | | | | | |
| 9 | | 1 | | | | | | | |
| 10 | | | | | | | | | |
| 11 | | | | | | | | | |
| 12 | | 2 | | | | | | | |
| >12 | | | | | | | | | |
| Total | 72 | 24 | 13 | 3 | 2 | 3 | 2 | 0 | 0 |

| No. of edit operations performed by recovery method | No. of prefix-defined errors | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 47 | 1 | 1 | | | | | | |
| 2 | 13 | 11 | 2 | | 1 | | | | |
| 3 | 3 | 5 | 5 | | | 1 | 1 | | |
| 4 | 2 | 3 | 2 | 1 | | | | | |
| 5 | 1 | 2 | 4 | 1 | 1 | | | | |
| 6 | | | 1 | | | | 2 | | |
| 7 | 1 | | | | | 2 | | | |
| 8 | | 1 | | | | | 1 | | |
| 9 | | | | | | | 1 | | |
| 10 | | | | | | | | | |
| 11 | | | | | | | | | |
| 12 | 1 | 1 | | | | | | | |
| >12 | | | | | | | | | |
| Total | 68 | 24 | 15 | 2 | 2 | 3 | 5 | 0 | 0 |

**Fig. C.5.** Numbers of programs, edit operations against errors:

Pascal, Recovery Method 2, $\sigma = 6$, $\tau = 12$.

| No. of edit operations performed by recovery method | No. of minimum-distance errors | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 46 | 2 | 1 | | | | | | |
| 2 | 14 | 12 | 3 | | | 1 | | | |
| 3 | 3 | 5 | 3 | 1 | | | | | |
| 4 | 3 | 3 | 1 | | | | | | |
| 5 | 2 | | 2 | 1 | 1 | | | | |
| 6 | | | 3 | | | 1 | 2 | | |
| 7 | 1 | | | | | 1 | | | |
| 8 | 1 | 1 | | 1 | | | | | |
| 9 | 1 | | | | | | | | |
| 10 | | | | | | | | | |
| 11 | | | | | | | | | |
| 12 | | | | | | | | | |
| >12 | 1 | 1 | | | 1 | | | | |
| Total | 72 | 24 | 13 | 3 | 2 | 3 | 2 | 0 | 0 |

| No. of edit operations performed by recovery method | No. of prefix-defined errors | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 46 | 2 | 1 | | | | | | |
| 2 | 13 | 11 | 5 | | | 1 | | | |
| 3 | 3 | 5 | 3 | 1 | | | | | |
| 4 | 3 | 3 | 1 | | | | | | |
| 5 | 1 | 1 | 1 | 1 | 1 | | 1 | | |
| 6 | | | 3 | | | 1 | 2 | | |
| 7 | | | 1 | | | 1 | | | |
| 8 | 1 | | | | | | 2 | | |
| 9 | | 1 | | | | | | | |
| 10 | | | | | | | | | |
| 11 | | | | | | | | | |
| 12 | | | | | | | | | |
| >12 | 1 | 1 | | | 1 | | | | |
| Total | 68 | 24 | 15 | 2 | 2 | 3 | 5 | 0 | 0 |

**Fig. C.6.** Numbers of programs, edit operations against errors:

Pascal, Recovery Method 2, $\sigma = 7$, $\tau = 14$.

| No. of edit operations performed by recovery method | No. of minimum-distance errors | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 208 | | | | | | | | |
| 2 | | 29 | | | | | | | |
| 3 | 3 | | 1 | | | | | | |
| 4 | 1 | 2 | | | | | | | |
| 5 | 1 | 6 | 2 | | 2 | | | | |
| 6 | | 2 | | | | | | | |
| 7 | | 2 | 4 | | | | | | |
| 8 | 7 | 2 | 1 | | | | | | |
| 9 | | | 1 | | | | | | |
| 10 | | | | | | | | | |
| 11 | | 1 | | | | | | | |
| 12 | | 2 | | | | | | | |
| >12 | 20 | 8 | 11 | 12 | 2 | 4 | 3 | | |
| Total | 240 | 54 | 20 | 12 | 4 | 4 | 3 | 0 | 0 |

| No. of edit operations performed by recovery method | No. of prefix-defined errors | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 208 | | | | | | | | |
| 2 | | 29 | | | | | | | |
| 3 | 3 | | 1 | | | | | | |
| 4 | 1 | 2 | | | | | | | |
| 5 | 1 | 6 | 2 | | 2 | | | | |
| 6 | | 2 | | | | | | | |
| 7 | | 2 | 4 | | | | | | |
| 8 | 5 | 4 | 1 | | | | | | |
| 9 | | | | | | | 1 | | |
| 10 | | | | | | | | | |
| 11 | | 1 | | | | | | | |
| 12 | | 2 | | | | | | | |
| >12 | 15 | 7 | 16 | 2 | 2 | 4 | 14 | | |
| Total | 233 | 55 | 24 | 2 | 4 | 4 | 15 | 0 | 0 |

**Fig. C.7.** Numbers of programs, edit operations against errors:

Pascal (weighted), Recovery Method 1, $\rho = 5$.

| No. of edit operations performed by recovery method | No. of minimum-distance errors | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 213 | | | | | | | | |
| 2 | | 5 | | | | | | | |
| 3 | 3 | 1 | | | | | | | |
| 4 | 1 | 3 | | | | | | | |
| 5 | 1 | 6 | 2 | | | | | | |
| 6 | | 3 | | | | | | | |
| 7 | | 2 | 1 | | | | | | |
| 8 | 3 | 2 | 1 | | | | | | |
| 9 | | 1 | | | | | | | |
| 10 | | | | | | | | | |
| 11 | | 1 | 3 | | | | | | |
| 12 | | 3 | | | | | | | |
| >12 | 19 | 27 | 13 | 12 | 4 | 4 | 3 | | |
| Total | 240 | 54 | 20 | 12 | 4 | 4 | 3 | 0 | 0 |

| No. of edit operations performed by recovery method | No. of prefix-defined errors | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 213 | | | | | | | | |
| 2 | | 5 | | | | | | | |
| 3 | 3 | 1 | | | | | | | |
| 4 | 1 | 3 | | | | | | | |
| 5 | 1 | 6 | 2 | | | | | | |
| 6 | | 3 | | | | | | | |
| 7 | | 2 | 1 | | | | | | |
| 8 | | 4 | 2 | | | | | | |
| 9 | | | 1 | | | | | | |
| 10 | | | | | | | | | |
| 11 | | 1 | 3 | | | | | | |
| 12 | | 3 | | | | | | | |
| >12 | 15 | 26 | 16 | 2 | 4 | 4 | 14 | | |
| Total | 233 | 55 | 24 | 2 | 4 | 4 | 15 | 0 | 0 |

**Fig. C.8.** Numbers of programs, edit operations against errors:

Pascal (weighted), Recovery Method 1, $\rho = 10$.

# Bibliography


AHO, A. V. 1980. "Translator writing systems: where do they now stand?", *IEEE Computer* **13:8**, 9-14.

AHO, A. V., D. S. HIRSCHBERG, AND J. D. ULLMAN. 1976. "Bounds on the complexity of the longest common subsequence problem", *J. ACM* **23:1**, 1-12.

AHO, A. V. AND S. C. JOHNSON. 1974. "LR parsing", *Computing Surveys* **6:2**, 99-124.

AHO, A. V., S. C. JOHNSON, AND J. D. ULLMAN. 1975. "Deterministic parsing of ambiguous grammars", *Comm. ACM* **18:8**, 441-452.

AHO, A. V., B. W. KERNIGHAN, AND P. J. WEINBERGER. 1988. *The Awk Programming Language*, Addison-Wesley, Reading, MA.

AHO, A. V. AND T. G. PETERSON. 1972. "A minimum-distance error-correcting parser for context-free languages", *SIAM J. Computing* **1:4**, 305-312.

AHO, A. V., R. SETHI, AND J. D. ULLMAN. 1986. *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA.

AHO, A. V. AND J. D. ULLMAN. 1972. *The Theory of Parsing, Translating and Compiling*, Vol. I: *Parsing*, Prentice-Hall, Englewood, NJ.

AHO, A. V. AND J. D. ULLMAN. 1977. *Principles of Compiler Design*, Addison-Wesley, Reading, MA.

AMMANN, U. 1978. "Error recovery in recursive descent parsers", *State of the Art and Future Trends in Compilation*, INRIA, Paris, 231-238.

ANDERSON, S. O. AND R. C. BACKHOUSE. 1981. "Locally least-cost error recovery in Earley's algorithm", *ACM TOPLAS* **3:3**, 318-347.

ANDERSON, S. O., R. C. BACKHOUSE, E. H. BUGGE, AND C. P. STIRLING. 1983. "An assessment of locally least-cost error recovery", *Computer J.* 26:1, 15-24.

BACKHOUSE, R. C. 1979. *Syntax of Programming Languages, Theory and Practice*, Prentice-Hall, London.

BACKHOUSE, R. C. 1984. "Global data flow analysis problems arising in locally least-cost

error recovery", *ACM TOPLAS* **6:2**, 192-214.

BARNARD, D. T. 1981. "Syntax error handling techniques", TR 81-125, Dept of Computing and Information Science, Queen's University, Kingston, Ontario.

BARNARD, D. T. AND R. C. HOLT. 1982. "Hierarchic syntax error repair for LR grammars", *International J. Computer and Information Sciences* **11:4**, 231-258.

BAUER, F. L. AND J. EICKEL. 1974. *Compiler Construction: An Advanced Course*, Springer-Verlag, Berlin.

BERTSCH, E. 1983. "Syntactic optimization of expressions - an experimental approach", Informatik Berichte Nr 39, Fern Universitat, Hagen.

BORNAT, R. 1979. *Understanding and Writing Compilers*, Macmillan, London.

BOUILLIER, P. 1977. "Automatic syntactic error recovery for LR parsers", *Implementation and Design of Algorithmic Languages, Proc 5th Annual III Conf*, Guidel, Roquencourt, INRIA, 1977.

BOUILLIER, P. AND M. JOURDAN. 1987. "A new error repair and recovery scheme for lexical and syntactic analysis", *Science of Computer Programming* **9**, 271-286.

BROWN, P. J. 1982. ""My system gives excellent error messages" - or does it", *Software - Practice and Experience* **12**, 91-94.

BROWN, P. J. 1983. "Error messages: the neglected area of the man/machine interface?", *Comm. ACM* **26:4**, 246-249.

BURGESS, C. AND L. JAMES. 1981. "An indexed bibliography for LR grammars and parsers", *ACM SIGPLAN Notices* **16:8**, 18-26.

BURKE, M. G. AND G. A. FISHER. 1982. "A practical method for syntactic error diagnosis and recovery", *ACM SIGPLAN Notices* **17:6**, 67-78.

BURKE, M. G. AND G. A. FISHER. 1987. "A practical method for LR and LL syntactic error diagnosis and recovery", *ACM TOPLAS* **9:2**, 164-197.

CHAPMAN, N. P. 1987. *LR Parsing: Theory and Practice*, Cambridge University Press, Cambridge.

CHYTIL, M. P. AND J. DEMNER. 1987. "Panic mode without panic", *Proc. 4th*

*International Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science, Springer-Verlag, Berlin.

CIESINGER, J. 1979. "A bibliography of error-handling", *ACM SIGPLAN Notices* **14:1**, 16-26.

CONWAY, R. W. AND W. L. MAXWELL. 1973. "CORC - the Cornell Computing Language", *Comm. ACM* **6:6**, 317-321.

CONWAY, R. W. AND T. R. WILCOX. 1973. "Design and implementation of a diagnostic compiler for PL/I", *Comm. ACM* **16:3**, 169-179.

COOPER, D. 1983. *Standard Pascal User Reference Manual,* Norton, New York.

CORBETT, R. P. 1985. "Static semantics and compiler error recovery", Report No. UCB/CSD 85/251, Computer Science Division (EECS), University of California, Berkeley, CA.

DAIN, J. A. 1984. "Error recovery schemes in LR parsers", Theory of Computation Report 71, University of Warwick, Coventry.

DAIN, J. A. 1985. "Error recovery for yacc parsers", *Proc. EUUG Autumn 1985 Conference*, Copenhagen, 159-166.

DAIN, J. A. 1987. "Minimum distance error correction", Research Report No. 102, University of Warwick, Coventry.

DENCKER, P. 1985. "Some topics in parser generation", IR-105, Vrije Universiteit, Amsterdam.

DEREMER, F. L. 1971. "Simple LR(k) grammars", *Comm. ACM* **14:7**, 453-460.

DEREMER, F. L. AND T. PENNELLO. 1982. "Efficient computation of LALR(1) look-ahead sets", *ACM TOPLAS* **4:4**, 615-649.

DION, B. A. 1982. "Locally least-cost error correctors for context-free and context-sensitive parsers", UMI Research Press, Ann Arbor, 1982.

DRUSEIKIS, F. C. AND G. D. RIPLEY. 1976. "Error recovery for simple LR(k) parsers", *Proc. ACM Annual Conf.*, 396-400.

DWYER, B. 1981. "A user-friendly algorithm", *Comm. ACM* **24:9**, 556-561.

EARLEY, J. 1970. "An efficient context-free parsing algorithm", *Comm. ACM* **13:2**, 94-102.

EHRENFEUCHT, A. AND D. HAUSSLER. 1986. "A new distance metric on strings computable in linear time", UCSC-CRL-86-27, Computer Research Laboratory, University of California, Santa Cruz, CA.

FEYCOCK, S. AND P. LAZARUS. 1976. "Syntax-directed correction of syntax errors", *Software - Practice and Experience* **6**, 207-219.

FISCHER, C. N. AND R. J. LEBLANC. 1988. *Crafting a Compiler*, Benjamin Cummings, Menlo Park, CA.

FISCHER, C. N. AND J. MAUNEY. 1980. "On the role of error productions in syntactic error correction", *Computer Languages* **5**, 131-139.

FISCHER, C. N., D. R. MILTON, AND S. B. QUIRING. 1980. "Efficient LL(1) error correction and recovery using only insertions", *Acta Inf.* **13**, 141-154.

FISCHER, C. N., K. C. TAI, AND D. R. MILTON. 1979. "Immediate error detection in strong LL(1) parsers", *Information Processing Letters* **8**, 261-266.

FISCHER, M. J. AND M. S. PATERSON. 1974. "String-matching and other products", *SIAM-AMS Procs* **7**, 113-125.

GANNON, J. D. AND J. J. HORNING. 1975. "Language design for programming reliability", *IEEE Trans. on Software Engineering* **SE-1:2**, 179-191.

GHEZZI, C. 1975. "LL(1) grammars supporting an efficient error handling", *Information Processing Letters* **3:6**, 174-176.

GINSBURG, S. AND N. LYNCH. 1976. "Size complexity in context-free grammar forms", *J. ACM* **23:4**, 582-598.

GRAHAM, S. L., M. A. HARRISON, AND W. L. RUZZO. 1980. "An improved context-free recognizer", *ACM TOPLAS* **2:3**, 415-462.

GRAHAM, S. L., C. B. HALEY, AND W. N. JOY. 1979. "Practical LR error recovery", *ACM SIGPLAN Notices* **14:8**, 168-175.

GRAHAM, S. L. AND S. P. RHODES. 1975. "Practical syntactic error recovery", *Comm.*

*ACM* **18:11**, 639-649.

GRAY, R. W. [undated manuscript]. "Generating fast, error recovering parsers", Computer Science Dept, University of Colorado, Boulder, CO.

GRAY, R. W. 1987. "Automatic error recovery in a fast parser", USENIX Procs, Phoenix, 337-346.

GREIBACH, S. 1968. "A note on undecidable properties of formal languages", *Mathematical Systems Theory* **2:1**, 1-6.

GRIES, D. 1971. *Compiler Construction for Digital Computers*, Wiley, New York, NY.

GRIES, D. 1974. "Error recovery and correction - an introduction to the literature", in Bauer and Eickel (1974), 627-638.

GROENING, K. AND C. OHSENDOTH. 1986. "NEMO - a nicely modified Yacc", *ACM SIGPLAN Notices* **21:4**, 58-66.

GRUNE, D. AND C. J. H. JACOBS. 1988. "A programmer-friendly LL(1) parser generator", *Software - Practice and Experience* **18:1**, 29-38.

HALL, P. A. V. AND G. R. DOWLING. 1980. "Approximate string matching", *Computing Surveys* **12:4**, 381-402.

HARPER, R., R. MILNER, AND M. TOFTE. 1988. "The definition of Standard ML", ECS-LFCS-88-62, LFCS, Dept of Computer Science, University of Edinburgh, Edinburgh.

HAMMOND K. AND V. J. RAYWARD-SMITH. 1984. "A survey on syntactic error recovery and repair", *Computer Languages* **9:1**, 51-67.

HARTMANN, A.C. 1977. *A Concurrent Pascal Compiler for Minicomputers*, Springer-Verlag, Berlin.

HEAPS, H. S. AND T. RADHAKRISHNAN. 1977. "Compaction of diagnostic messages for compilers", *Software - Practice and Experience* **7:1**, 139-144 .

HOPCROFT, J. E. AND J. D. ULLMAN. 1966. "Error correction for formal languages", Digital System Laboratory Technical Report 52, Princeton University, Princeton, NJ.

HOPCROFT, J. E. AND J. D. ULLMAN. 1979. *Introduction to Automata Theory,*

*Languages, and Computation.* Addison-Wesley, Reading, MA.

HORNING, J.J. 1974. "What the compiler should tell the user", in Bauer and Eickel (1974), 525-548.

HUNTER, R. 1981. *The Design and Construction of Compilers*, Wiley, New York, NY.

IRONS, E.T. 1963. "An error-correcting parse algorithm", *Comm. ACM* **6:11**, 669-673.

JAMES, L.R. 1972. "A syntax directed error recovery method", Technical Report CSRG-13, Computer Systems Research Group, University of Toronto, Toronto.

JOHNSON, S. C. 1975. "Yacc - yet another compiler-compiler", Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ.

JOHNSON, S. C. 1978. "A portable compiler - theory and practice", *Conf. Record Fifth Annual ACM Symposium on Principles of Programming Languages*, ACM, New York, NY, 97-104.

JOHNSON, S. C. 1988. "Yacc meets C++", *Procs. EUUG Spring Meeting*, London, 53-57.

KANTOROWITZ, E. AND H. LAOR. 1986. "Automatic generation of useful syntax error messages", *Software - Practice and Experience* **16:7**, 627-640.

KIEBURTZ, R.B. AND A. B. PAI. 1979. "Global context recovery: a new strategy for parser recovery from syntax errors", *ACM SIGPLAN Notices* **14:8**, 158-167.

KNUTH, D.E. 1965. "On the translation of languages from left to right", *Information and Control* **8:6**, 607-639.

KNUTH, D.E. 1968. "Semantics of context-free languages", *Mathematical Systems Theory* **2:2**, 127-145.

KOSKIMIES, K., T. ELOMAA, T. LEHTONEN, AND J. PAAKKI. 1988. "TOOLS/HLP84 report and user manual", Report A-1988-2, Dept of Computer Science, University of Helsinki, Helsinki.

KOSKIEMIS, K., O. NURMI, J. PAAKKI, AND S. SIPPU. "The design of a language processor generator", *Software - Practice and Experience* **18:2**, 107-135.

KRAWCZYK, T. 1980. "Error correction by mutational grammars", *Information Processing Letters* **11:1**, 9-15.

KRUSKAL, J. B. AND D. SANKOFF (Eds). 1983. *Theory and Practice of Sequence Comparison: Tme Warps, String Edits, and Macromolecules,* Addison-Wesley, Reading, MA.

KRUSKAL, J. B. 1983. "An overview of sequence comparison: time warps, string edits, and macromolecules", *SIAM Review* **25:2**, 201-237.

LEINIUS, R. P. 1970. "Error detection and recovery for syntax directed systems", Ph. D. Thesis, Computer Science Department, University of Wisconsin, Madison.

LESK, M. E. 1975. "Lex - a lexical analyzer generator", Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ.

LEVY, J.P. 1975. "Automatic correction of syntax errors in programming languages", *Acta Informatica* **4:3**, 271-292.

LEWI, J., K. DE VLAMINCK, J. HUENS, AND M. HUYBRECHTS. 1978. "The ELL(1) parser generator and the error recovery mechanism", *Acta Informatica* **10**, 209-228.

LOWRANCE, R. AND R. A. WAGNER. 1975. "An extension of the string-to-string correction problem", *J. ACM* **22:2**, 177-183.

LYON, G. 1974. "Syntax-directed least-errors analysis for context-free languages: a practical approach", *Comm. ACM* **17:1**, 3-14.

MCKEEMAN, V. M., J. J. HORNING, AND D. B. WORTMAN. 1970. *A Compiler Generator*, Prentice-Hall, Englewood Cliffs, NJ.

MASEK, W. J. AND M. S. PATERSON. 1980. "A faster algorithm computing string edit distances", *J. Computer and Systems Sciences* **20**, 18-31.

MATHIES, T. 1988. "A fast parallel algorithm to determine edit distance", Carnegie Mellon Computer Science Technical Report CMU-CS-88-130, Carnegie Mellon University, Pittsburgh, PA.

MATTERN, F. 1984. "An annotated bibliography on error handling in compilers", Sonderforschungsbereich 124, "VLSI und Parallelitat", Teilprojekt D1, Bericht Nr 18/84, Fachbereich Informatik, Universitat Kaiserslautern, Kaiserslautern.

MAUNEY, J. AND C. N. FISCHER. 1981. "An improvement to immediate error detection in

strong LL(1) parsers", *Information Processing Letters* **12:5**, 211-212.

MAUNEY, J. AND C. N. FISCHER. 1982. "A forward move algorithm for LL and LR Parsers", *ACM SIGPLAN Notices* **17:6**, 79-87.

MAUNEY, J. AND C. N. FISCHER. 1988. "Determining the extent of lookahead in syntactic error repair", *ACM TOPLAS* **10:3**, 456-469.

MICKUNAS, M.D. AND J. A. MODRY. 1978. "Automatic error recovery for LR parsers", *Comm. ACM* **21:6**, 459-465.

MOITRA, A. AND M. JOSEPH. 1989. "Implementing real-time systems by transformation", Dept. of Computer Science, University of Warwick.

MORGAN, H.L. 1970. "Spelling corrections in systems programs", *Comm. ACM* **13:2**, 90-94.

MOULTON, P.G. AND M. E. MULLER. 1967. "DITRAN - a compiler emphasizing diagnostics", *Comm. ACM* **10:1**, 45-52.

NORMAN, D. A. 1983. "Design rules based on analyses of human error", *Comm. ACM* **26:4**, 255-258.

OTTMANN, T. (Ed). 1987. *Fourteenth International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, Springer-Verlag, Berlin.

OWOLABI, O. AND D. R. MCGREGOR. "Fast approximate string matching", *Software - Practice and Experience* **18:4**, 387-393.

PAI, A. AND R. B. KIEBURTZ. 1979. "Global context recovery: a new strategy", *ACM SIGPLAN Notices* **14:8**, 158-167.

PAI, A. AND R. B. KIEBURTZ. 1980. "Global context recovery: a new strategy for syntactic error recovery by table-driven parsers", *ACM TOPLAS* **2:1**, 18-41.

PEMBERTON, S. 1980. "Comments on an error recovery scheme by Hartmann", *Software - Practice and Experience* **10**, 231-240.

PENNELLO, T.M. AND F. DEREMER. 1978. "A forward move algorithm for LR error recovery", *Conf. Record Fifth Annual ACM Symposium on Principles of Programming Languages*, 241-254.

PETERSON, J. L. 1980. "Computer programs for detecting and correcting spelling errors", *Comm. ACM* **23:12**, 676-687.

PETERSON, T. G. 1972. "Syntax error detection, correction and recovery in compilers", Ph. D. thesis, Stevens Institute of Technology.

POONEN, G. 1977. "Error recovery for LR(k) parsers", *Information Processing 77*, North Holland Publishing Co., 529-533.

POST, E. 1946. "A variant of a recursively unsolvable problem", *Bull. AMS* **52**, 264-268.

PRUSINKIEWICZ, P. 1976. "Error detection using regular languages", *Procs. Fifth Symposium on Mathematical Foundations of Computer Science*. Lecture Notes in Computer Science, Springer-Verlag, Berlin, 458-464.

RAIHA, K. J. 1980. "Experiences with the compiler writing system HLP", *Procs. of a Workshop on Semantics-Directed Compiler Generation,* Lecture Notes in Computer Science, Springer-Verlag, Berlin, 350-362.

RAIHA, K. J., M. SAARINEN, M. SARJAKOSKI, S. SIPPU, E. SOISALON-SOININEN, AND M. TIENARI. 1983. "Revised report on the compiler writing system HLP78", Report A-1983-1, Dept of Computer Science, University of Helsinki, Helsinki.

REPS, T. W. AND T. TEITELBAUM. 1989. *The Synthesizer Generator - A System for Constructing Language-based Editors,* Springer-Verlag, New York, NY.

RIPLEY, D.C. AND F. C. DRUSEIKIS. 1978. "A statistical analysis of syntax errors", *Computer Languages* **3**, 227-240.

ROEHRICH, J. 1980. "Methods for the automatic construction of error correcting parsers", *Acta Inf.* **13:2**, 115-139.

SALOMAA, A. AND M. STEINBY. 1977. *Fourth Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, Springer-Verlag, Heidelberg.

SHNEIDERMAN, B. 1982. "Designing computer system messages", *Comm. ACM* **25:9**, 610-611.

SIPPU, S. 1981. "Syntax error handling in compilers", Report A-1981-1, Dept of Computer Science, University of Helsinki, Helsinki.

SIPPU, S. AND E. SOISALON-SOININEN. 1977. "On defining error recovery in context-free parsing", in Salomaa and Steinby (1977), 492-503.

SIPPU, S. AND E. SOISALON-SOININEN. 1980. "A scheme for LR(k) parsing with error recovery. Part 1: LR(k) parsing. Part 2: error recovery. Part 3: error correction", *Int. J. Computer Math.* **8**, 27-42, 107-119, 189-206.

SIPPU, S. AND E. SOISALON-SOININEN. 1982. "Practical error recovery in LR parsing", *Conf. Record Ninth Annual ACM Symposium on Principles of Programming Languages*, ACM, New York, NY, 177-184.

SIPPU, S. AND E. SOISALON-SOININEN. 1983. "A syntax-error handling technique and its experimental analysis", *ACM TOPLAS* **5:4**, 656-679.

SPENKE, M., H. MUHLENBEIN, M. MEVENKAMP, F. MATTERN, AND C. BEILKEN. 1984. "A language independent error recovery method for LL(1) parsers", *Software - Practice and Experience* **14:11**, 1095-1107.

STIRLING, C. 1985. "Follow set error recovery", *Software - Practice and Experience* **15:3**, 239-257.

STROUSTRUP, B. 1986. *The C++ Programming Language*, Addison-Wesley, Reading, MA.

TAI, K.C. 1978. "Syntactic error correction in programming languages", *IEEE Trans. on Software Engineering* **SE-4:5**, 414-423.

TAI, K.C. 1980a. "Locally minimum-distance correction of syntax errors in programming languages", *Procs. ACM National Conference*, ACM, New York, NY, 204-210.

TAI, K.C. 1980b. "Predictors of context-free grammars", *SIAM J. Comput.* **9:3**, 653-664.

TANENBAUM, A. S., H. VAN STAVAREN, E. G. KEIZER, AND J. W. STEVENSON. 1983. "A practical toolkit for making portable compilers", *Comm. ACM* **26:9**, 654-660.

TEITELBAUM, R. 1975. "Minimal distance analysis of syntax errors in computer programs", Ph. D. thesis, Carnegie Mellon University, Pittsburgh, PA.

TURNER, D. A. 1977. "Error diagnosis and recovery in one pass compilers", *Information Processing Letters* **6:4**, 113-115.

UKKONEN, E. 1985. "Upper bounds on the size of LR(k) parsers", *Information Processing Letters* **20:2**, 99-103.

UNITED STATES DEPT OF DEFENSE. 1981. *The Programming Language Ada Reference Manual*", Lecture Notes in Computer Science, Springer-Verlag, Berlin.

WAGNER, R. 1974. "Order-n correction for regular languages", *Comm. ACM* **17:5**, 265-268.

WAGNER, R. A. AND M. J. FISCHER. 1974. "The string-to-string correction problem", *J. ACM* **21:1**, 168-173.

WETHERELL, C. 1977. "Why automatic error correctors fail", *Computer Languages* **2**, 179-186.

WETHERELL, C. AND A. SHANNON. 1981. "LR - automatic parser generator and LR(1) parser", *IEEE Trans. on Software Engineering* **SE-7:3**, 274-278.

WIRTH, N. 1968. "PL360 - a programming language for the 360 computers", *J. ACM* **15:1**, 37-74.

WIRTH, N. 1971. "Design of a Pascal compiler", *Software - Practice and Experience* **1**, 309-333.

WIRTH, N. 1976. *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

WONG, C. K. AND A. K. CHANDRA. 1976. "Bounds for the string editing problem", *J. ACM* **23:1**, 13-16.