INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

University Microfilms International
300 North Zeeb Road
Ann Arbor, Michigan 48106 USA
St. John's Road, Tyler's Green
High Wycombe, Bucks, England HP10 8HR
SHIELDS, THOMAS EVANS
SYNTAX DIRECTED ERROR ANALYSIS IN AUTOMATICALLY CONSTRUCTED PARSERS.

RICE UNIVERSITY, PH.D., 1976
RICE UNIVERSITY

SYNTAX DIRECTED ERROR ANALYSIS
IN
AUTOMATICALLY CONSTRUCTED PARSERS

by

Thomas E. Shields

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

THESIS DIRECTOR'S SIGNATURE:

Kenneth W. Kennedy Jr.

HOUSTON, TEXAS

MAY, 1978
ACKNOWLEDGEMENTS

I would like to thank my advisor, Ken Kennedy, for his suggestions and encouragement, Scott Warren for many fruitful discussions, and Sharon Parker, whose assistance made the words which follow a part of reality. This manuscript is dedicated to those members of The Family who believed in me, lest I forget.
# TABLE OF CONTENTS

0. Preliminaries.  
0.1 Introduction. 1  
0.2 Desiderata. 5  
0.3 Notation and Definitions. 7  

1. General Preview. 10  
1.1 Background perspective. 10  
1.2 Overview of research. 20  

2. Review of Related Research 25  
2.1 Introduction 25  
2.2 Local strategies. 26  
2.3 Global strategies. 44  
2.4 Review. 48  

3. Error Detection Without Recovery. 50  
3.1 Motivation. 50  
3.2 The error detection model. 57  
3.3 Extended parsing. 73  
3.4 Extended parsing in perspective. 93  

4. Efficient Error Detection. 95  
4.1 Introduction. 95  
4.2 Context Condensation. 96  
4.3 Extended LR(k) parsing. 112  
4.4 The size of an extended LR(k) parser. 132  
4.5 Resolution of nondeterminism. 134  
4.6 Optimizations. 147  

5. Summary and Conclusions. 148  
5.1 Summary. 148  
5.2 Limitations. 152  
5.3 Significance of this work. 154  
5.4 Recommendations for further research. 157  

Appendix I. An SLR(1) parser. 159  
Appendix II. An extended SLR(1) parser. 166  
References. 176
Chapter 0
PRELIMINARIES

0.1 Introduction
The static structure of a programming language, by which is meant the structure of a well-formed program apart from its intended operational meaning, has three essential levels of increasing complexity.

1. The lexical structure is the lowest level normally considered in specifying a programming language, and is often modeled by the regular languages. This level specifies how the primitive atoms, usually referred to as characters, may be composed into sequences forming symbols of the higher levels, and how these symbols may be separated by other sequences of characters (typographical display features[61]) during the drafting of a particular program.

2. The syntactic structure (or syntax) determines rules of composition by which symbols may be combined into patterns, without regard to any meaning attached to a particular symbol. This level is generally modeled by the context-free languages, but often may be partially specified in terms of regular languages [35].
3. The **semantic** structure of a programming language defines the valid interactions among symbols in a pattern defined by the language's syntax. These interactions are based on inherent meanings attributed to certain symbols, the context-sensitive properties of the language.

Each of these levels is subject to the introduction of errors, so a compiler for any programming language must be prepared to cope with this eventuality. As noted by Horning [24], "... most of the programs processed by any compiler will be to some degree incorrect ... simply because correct programs need not be recompiled ...".

This dissertation reports research in automatic construction of compilers with syntax-directed error analysis capabilities, the term "syntax-directed" referring to dependency on a formal specification of the syntax of a particular programming language. Analysis of syntax errors is difficult even with ad hoc approaches due to the far-ranging effects a particular error may have on the program as seen by a parser. For example, omitting a single **begin** or **end** in an Algol-like language will often drastically alter the interpretation of a large section of the input being compiled. Lexical or semantic errors, on the other hand, typically have relatively local effects [24].
One of the primary practical goals of research into the theory of formal languages is the automatic generation of that part of a compiler (usually referred to as the front end) involved in verifying structural correctness of a particular program (recognition) and communicating the structure of that program to the phases concerned with translation or code generation. Thus far, models of the recognition process have been developed which enable the automatic construction of efficient regular and context-free language recognizers. This has provided the means for partially automating the construction of the lexical and syntactic analysis phases of a compiler. Extending the recognition process to compute the parse of a particular sentence solves the basic communication problem between the syntactic and the semantic analysis phases.

Unfortunately, language recognition is not a particularly good model of the process required in a practical compiler. A reconizer can only reject a string which is not a member of some specific language. If the string was intended to be a member of that language, then a simple rejection of the input is not an acceptable response. The minimal requirement is that the compiler explicitly identify the error which caused the rejection.
Such minimal behavior might be adequate for an interactive, (incremental) compiling environment (see [58]), but in a batch system it is essential that all errors (or as many as possible) be reported - not just the first. This requirement has resulted in the development of various ad hoc strategies which extend the language recognition process to detect and report additional errors.

The dissertation is organized as follows. The remainder of the current chapter is devoted to a discussion of the desiderata of automatic syntax error analysis, followed by a review of basic notation and some preliminary definitions. Chapter 1 presents a general review of background material and a brief overview of the principle results of the research. Chapter 2 contains a review of related research in the area of syntax-directed error recovery and correction. Chapter 3 presents a model of the language recognition problem which includes the notion of error detection as a integral concept. Chapter 4 describes an approach to the construction of efficient error detecting parsers based on the model developed in Chapter 3. Summary and conclusions appear in Chapter 5.
0.2 Desiderata

The basic goal of syntax error analysis is that the compiler be able to deal with any input string in a reasonable way. Even aborting compilation because of the density of errors is acceptable if accompanied by a reasonable explanation. Such problems as empty input, unexpected end of input, and extra input after the anticipated end of input should be handled uniformly by the same mechanism.

Perhaps the principle characteristic desired of truly automatic syntax-directed error analysis is that it not depend on additional specification beyond that required for generation of a parser. That is, the only input should be the grammar used to specify the language. Otherwise, detailed knowledge of how the performance is affected by the additional information is necessary to successfully "program" the error analyzer. Although it is certainly desirable to give the implementor the option of "fine tuning" the analyzer, successful performance should not depend on it.

The second crucial property is that detection of an error should not adversely affect the analyzer's ability to detect additional errors, or the processing of correct programs. Rhodes [51] suggests that "a good error detection ... scheme should maximize the number of errors detected, but minimize the number of times it reports an error when there is none."
Since it is difficult for the compiler to know where the exact error is, "detection of an error" is realistically interpreted as "detection of the existence of an error." Levy [35] demonstrated that the distance between the point of detection and minimum distance correction may be unbounded. In order to maximize the potential for correct interpretation of the programmer's intentions, the error analyzer should attempt to localize errors to as small an interval as possible. This allows a diagnosis to be more specific about what is wrong and implies that the error analyzer should skip (ignore) as little of the input as possible.

One further desirable property is that the syntax error analyzer should enable the execution of the semantic analyzer on an incorrect program. This allows the compiler to discover semantic errors in the program, thereby enhancing the productivity of a single compilation. Thus the syntactic phase of a compiler should be, in the words of Horning [24], an error sink, i.e., syntax errors should be repaired to produce a syntactically correct program for the semantic phase. Additionally, the semantic phase must be made aware of the extent of a repair so that actual semantic errors can be distinguished from errors induced by the repairs. It would also be useful to allow local semantic analysis on correct segments of the compiler to assist in the syntactic error analysis.
0.3 Notation and Definitions

The reader is assumed to be familiar with the concepts of formal language theory and in particular with the concepts and constructions of \( LR(k) \) parsing, as presented in Aho and Ullman [3]. A context-free grammar is denoted \( G = (N, \Sigma, P, S) \) and sometimes also \( G = (N, \Sigma, P, S) \), where \( N \) is the nonterminal alphabet, \( \Sigma \) is the terminal alphabet, \( P \) is the production set and \( S \) (or \( S \)) is the distinguished sentence prototype of the grammar. Productions are usually denoted \( (A, \alpha) \) where \( A \) is the left side and \( \alpha \) is the right side; the production \( (A, \alpha) \) is sometimes represented as \( A \rightarrow \alpha \). Whenever the sets \( N \) and \( \Sigma \) are defined, the vocabulary of the context-free grammar is denoted by \( V \) and is always equivalent to \( N \cup \Sigma \); superscripts or subscripts on \( V \) always carry over to \( N \cup \Sigma \), e.g. \( V' \equiv N' \cup \Sigma' \). The term grammar means context-free grammar in this dissertation, although in many cases application to more general grammars is straightforward; the term language means context-free language. The relation derives is denoted \( \Rightarrow \), with the transitive and transitive-reflexive closures denoted \( \Rightarrow^+ \) and \( \Rightarrow^* \) respectively. A right derivation is denoted by \( \Rightarrow L_k^+ \) (\( \Rightarrow L_k^* \)).

Unless otherwise specified, the following conventions apply, regardless of the presence or absence of superscripts and subscripts:

(1) Terminal symbols are denoted by lower case letters at the beginning of the Latin alphabet \( (a, b, c, \ldots) \).
(2) Nonterminal symbols are denoted by upper case letters at the beginning of the Latin alphabet (A, B, C, ...), S, and Z.

(3) Strings over Σ are denoted by lower case letters at the end of the Latin alphabet (...x, y, z).

(4) Symbols in V are denoted by upper case letters at the end of the Latin alphabet excluding S and Z (...W, X, Y).

(5) Strings over V are denoted by lower case letters at the beginning of the Greek alphabet, excluding ε, (α, β, γ,...).

(6) The empty string is denoted by ε.

The end of a proof or algorithm is denoted by //.

If σ denotes any string of symbols, then

(1) lg (σ) denotes the length of the sequence of symbols represented by σ.

(2) σ^R denotes the reversal of σ.

(3) σ[i] for 1 ≤ i ≤ lg(σ), denotes the i-th symbol of σ; if i < 1 or i > lg(σ), then σ[i] denotes ε.

(4) σ[i:j], for 1 ≤ i ≤ j ≤ lg(σ), denotes the substring of σ composed of the i-th through the j-th symbols of σ; if i < 1, then σ[i:j] = σ[1:j]; if j > lg(σ), then σ[i:j] = σ[i:lg(σ)]; if i > j, then σ[i:j] denotes ε.

(5) a prefix of σ is any σ[1:j], for 1 ≤ j ≤ lg(σ); σ[1:j] is a proper prefix if j < lg(σ).
(6) a suffix of $\sigma$ is any $\sigma[i:\lfloor \lg(\sigma) \rfloor]$, for $1 \leq i \leq \lfloor \lg(\sigma) \rfloor$; $\sigma[i:\lfloor \lg(\sigma) \rfloor]$ is a proper suffix if $i > 1$.

(7) an infix of $\sigma$ is any $\sigma[i:j]$, for $1 \leq i \leq j \leq \lfloor \lg(\sigma) \rfloor$; $\sigma[i:j]$ is a proper infix if $1 < i < j < \lfloor \lg(\sigma) \rfloor$.

Unless otherwise specified, the term prefix (suffix, infix) refers to proper prefix (proper suffix, proper infix).

**Definition 0.1**

Let $G = (N, \Sigma, P, \mathcal{Z})$ be a context-free grammar. $\gamma \in V^*$ is a valid prefix of $G$ if there exist $\alpha, \beta_1, \beta_2 \in V^*$, $\mathcal{A} \in N$, $\mathcal{W} \in \Sigma^*$ such that $\mathcal{Z} \xrightarrow{\mathcal{F}} \sigma \mathcal{A} \mathcal{W} \xrightarrow{\mathcal{F}} \alpha \beta_1 \beta_2 \mathcal{W}$ and $\gamma = \alpha \beta_1$ [19].

**Definition 0.2**

In the algorithm for construction of a set of valid LR(k) items, the subset of an item set computed before the closure operation is performed is referred to as the basis of the item set.
Chapter 1
GENERAL PREVIEW

1.1 Background perspective

From the point of view of the syntactic analysis phase of a compiler, a program (referring to an input string, whether correct or not) is a linear pattern of atomic symbols. The set of all syntactically correct programs constitutes the particular language $L$ accepted by the syntax analyzer. The phase "syntactically correct" usually implies that a particular pattern of symbols conforms to the rules of pattern generation given by a context-free grammar $G$. A program is syntactically incorrect if and only if the particular pattern of symbols is not a member of the language $L$.

The formal approaches to syntactic error analysis are based on the work of Hamming [22] and model errors by transformations on correct strings. The primitive transformations are the simple edit operations of replacement, insertion, or deletion of a single symbol [23]. The basic model is the following.

Definition 1.1

An edit operator (or simply edit) $T_{i,a}$ is a function from $\Sigma^*$ into $\Sigma^*$ as follows:
(1) replacement
\[ R_{i,a}(x)_i = y \text{ iff for some } u,v \in \Sigma^* \text{ and } a,b \in \Sigma \text{ such that } \lg(u) = i \text{ and } a \neq b, x = ubv \text{ and } y = uav. \]

(2) insertion
\[ I_{i,a}(x)_i = y \text{ iff for some } u,v \in \Sigma^* \text{ and } a \in \Sigma \text{ such that } \lg(u) = i, x = uv \text{ and } y = uav. \]

(3) deletion
\[ D_{i,a}(x)_i = y \text{ iff for some } u,v \in \Sigma^* \text{ and } a \in \Sigma \text{ such that } \lg(u) = i, x = uav \text{ and } y = uv. \]

Definition 1.2
A string \( y \) in some language \( L \) is said to be an interpretation of \( x \in \Sigma^* \) with respect to \( L \) for \( x \notin L \) if \( x \) can be derived from \( y \) by a sequence of edits.

Definition 1.3
An error correction algorithm for a language \( L \) is an algorithm that implements a mapping from \( \Sigma^* \) into \( L \) such that a string \( x \notin L \) is mapped onto the interpretation of \( x \) for which \( \hat{x} \) "stands".

Ideally, the interpretation which a string \( x \) stands for is the string which the programmer intended to give to the compiler when instead he or she supplied \( x \). A compiler cannot practically be expected to know what the programmer intended.
Therefore, minimum distance, in terms of the number of edits required to transform an interpretation into the particular input string, is often assumed to be a reasonable measure of what the programmer intended. It is also possible to assign a weight to each particular edit operation.

**Definition 1.4**

Let $d(u,v)$ denote the (weighted) distance between $u$ and $v$ in terms of the least (weighted) number of edits required to transform $u$ into $v$. Then a (weighted) minimum distance error correction algorithm for a language $L$ is an error correction algorithm such that a string $x \notin L$ is mapped onto an interpretation $y \in L$ such that the (weighted) distance between $x$ and $y$ is minimal.

Notationally

$$d(x,y) = \min_{z \in L} d(x,z).$$

If the number of edits required is $k$, then $x$ is said to contain $k$ minimum distance errors.

A semantic difficulty with this model is the usage of the term "correction", which implies replacement of the incorrect input by the program which the programmer meant. "True error correction...is substantially beyond the current state of the art..." [24] in both programming language theory and artificial intelligence.
Usage of this term is often misleading, so the less suggestive term "reparation" is used in its place for the rest of this dissertation.

A more fundamental practical difficulty is that the location of errors in a string must be defined a posteriori by the edits in the sequence chosen to transform the minimum distance interpretation into the input string. The problem is that there may be several equal length transformation sequences for a particular interpretation, and there may be several different interpretations requiring the same number of edits. Therefore, the basic model provides a rather ambiguous definition of the errors in an incorrect program.

Furthermore, this type of model results in rather expensive algorithms. For general context-free languages, the worst-case bound on the time required for a general minimum distance analysis is \( O(n^3) \) for a string of length \( n \) [57], regardless of whether the string is correct or not. The result is not currently accepted as a practical one in the literature.

Most practical models of an incorrect program restrict the problem domain by assuming that the program does not have too many errors (for example, see [32, 35, 36, 56, 62]). Intuitively, there is practical justification for this, since even if a program contains errors it should still have some local resemblance to a correct program.
This seems to be further justified by empirical evidence [46,64]. The problem with this restriction is that the reparation process is likely to be faced, at some point, with a string which it cannot repair. This necessitates introduction of the notion of error recovery, and often results in treating reparation as an heuristic function which attempts to guide a parser-oriented recovery algorithm in an "intelligent" direction.

**Definition 1.5**

An error configuration of a parser $P$ is some configuration of $P$ in which there is no next move defined for $P$. The location in an input string $x$ corresponding to an error configuration reached by $P$ after processing some prefix of $x$ is called the parser defined error in $x$.

**Definition 1.6**

An error recovery algorithm for a parser $P$ is an algorithm that implements a mapping from the set of accessible error configurations of $P$ to the set of accessible non-error configurations of $P$. In any system in which a parser $P$ has been augmented by recovery capabilities, $P$ is referred to as the base parser.
The most common practical approach to syntax error analysis has been to design an ad hoc error recovery process directly around specific characteristics of the particular programming language. Such an approach is easily grafted onto any parsing algorithm, which is one reason for its popularity. The basic idea is to identify sequences of symbols which commonly occur in sentences and which uniquely specify what may follow the particular sequence (the simple case of a single symbol is the most used). Upon encountering an error, the recovery strategy is then to look for one of these sequences after the error point and resume parsing with what follows. Schneider [53] refers to these characteristic locations as synchronizable encodings, or synchronizing points in the simple case.

The drawbacks to this approach are due to its ad hocness. It is not generally possible to minimize the extent of the input string treated as incorrect after an error, since the method depends on predefined synchronizing points. Furthermore, it applies only to the particular language, so that an implementor must come up with a new set of recovery actions with each new language.

Levy [35], among others, states that any practical method should only require an amount of work proportional to the number of errors in the input string. In particular, a correct string should be processed in the same time as needed to parse the string, or at least not significantly slower. This has led to an approach whereby a specific
parsing technique is augmented with a recovery algorithm based on characteristics of the parsing technique. The recovery procedure usually operates as a subroutine of the parser. Whenever the parser detects an error, control is passed to the recovery procedure, which alters the parser's configuration in order to resynchronize the parser with the input, "repairing" the string locally around the error detection point. Control is then returned to the parser, which is (ideally) then able to continue forward to detect additional errors.

Justifications for this "bolt-on" approach are that such systems are easier to construct and comprehend, and tend to operate faster, making it much more suitable for use in a practical compiler than the minimum distance techniques. It trivially satisfies the goal that correct strings be processed in the same time as for parsing, since the recovery procedure is never invoked in this case. Furthermore, recovery is independent of the particular language, which makes this approach suitable for use in a parser generator system.

The limitation of this approach is its "one-way" relationship with the parser. The recovery process requires an intimate knowledge of the particular parsing technique in order to effect successful continuation of the parser. However, the only knowledge the parser has available is that, in the event an error is encountered, the recovery procedure is to be executed.
No details of the recovery actions are made available (or could be used if made available) to the parser, and the event itself is immediately forgotten (this does not necessarily apply to parsers with ad hoc extensions). The result is that it is difficult or impossible to prevent recovery from affecting the detection of subsequent errors.

The recovery-based approach to syntax error analysis leaves something to be desired in its current state of development. The error analysis is forced into a left-to-right definition and treatment of the errors in a string, as defined by a particular parsing algorithm. The location of a parser-defined error may not correspond in a simple way to the location of the actual error (or errors) made by the programmer, so that diagnostics generated during recovery may not indicate the true nature of the problem. Furthermore, each error must be resolved before the parser can continue, and only limited right context (in a left-to-right parser) is usually available during recovery. Thus, a bad recovery decision based on local context may cause the parser to detect an arbitrarily long cascade of spurious errors later, which is undesirable.

Fully automatic generation of error reparation capabilities for a compiler is a problem which could well be the subject of artificial intelligence research. Previous approaches to this problem are seen to be primarily attempts at heuristic solutions to a context-sensitive and possibly
intrinsically nondeterministic problem. This is exactly what researchers in artificial intelligence have been doing for years. The significance of this observation is that reasonable results are probably going to be realized only through a combination of algorithmic techniques and heuristic strategies based on knowledge about a specific problem domain. Examples of achievements in artificial intelligence point this out. Through use of specific knowledge, general heuristic methods can be tailored to a specific problem with remarkable results.

Error analysis in the domain of programming languages has close parallels with the problems of natural language understanding. "Natural" programming languages provide an advantage, however, over the more general problem because of their much simplified and restricted structure, both syntactically and semantically. Formal methods are available to model a particular programming language with much more precision than can be had with, say, English. Many of the problems requiring heuristic solutions for the general case are therefore solvable algorithmically, and those which require heuristic approaches for efficiency are at least given a precise formal structure from which to derive appropriate heuristics.

The successes of various hand-tailored "error-correcting" compilers, such as PL/C \([g]\), suggests that the appropriate problem domain with which to approach automatic
generation of error analyzers is a particular programming language. Most of the ad hoc automatic recovery strategies take this approach, but suffer in performance from a lack of sufficiently powerful heuristics. On the other hand, many of the more formally based automatic strategies, utilizing more powerful approaches, tend to suffer from a lack of specific knowledge about the particular language, relying instead on the structure of the grammar and the properties of the parsing technique.

These latter techniques usually handle this deficiency by allowing (more often requiring) the implementor to tailor the performance by hand for the particular language/grammar combination. Typical methods involve choosing appropriate productions, or supplying such additional information as the cost of deleting, inserting, or replacing symbols of the grammar in a sentential form for use in some pattern-matching process.
1.2 Overview of research

The approach to syntactic error analysis proposed in this dissertation seeks to combine the best features of the "practical" approaches with the global perspective found in the minimum distance approach. Correct strings can be analyzed in the same time as required for parsing, while incorrect strings can be analyzed globally only to the extent that errors are present. Locally correct segments of a program surrounding an error point can provide extended context for the analysis of the error. These capabilities require that the process of reparation be separated from error detection, which is accomplished by "eliminating" the need for error repair to detect errors in a string after the first. This approach was suggested, in part, by an ad hoc parsing strategy proposed by Schwartz [54], referred to as "eclectic parsing". His method was, basically, to construct a two-pass parser. The first pass is able to process any string of symbols and assign some arbitrary "derivation" tree structure to it. The second pass is then allowed to restructure this tree as desired to ease further semantic processing. Unexpected structure due to corrupted input is handled in an ad hoc but systematic way to generate error diagnostics.

The separation of error reparation from detection is further motivated by considering Bellman's dynamic programming techniques [5], which approaches optimization problems on a "divide and conquer" basis. By separating a function
to be optimized into independent segments, each segment may be individually optimized, allowing the construction of an optimal solution to the original function. Reparation obviously depends on the detection of errors in a string, however, the process of detecting errors should be independent of reparation in order to prevent the repair of one error from influencing the detection of additional errors, and to prevent the order of detection of errors from influencing the reparation. This suggests a serial decomposition of the syntactic error analysis problem into detection followed by reparation.

In order to achieve this decoupling, a model of syntax-directed parsing is proposed, suggested in part by the work of Druseikis and Ripley [13]. This model, referred to as extended parsing (after [13]), "merges" the processes of recognition, error detection and recovery. Extended parsing here is oriented toward the recognition and reduction of fragments of sentential forms rather than entire sentential forms. In this model an input string is assumed to consist of a sequence of fragments of sentences, so that sentence recognition becomes a special case of a more general process.

In the framework of this model a program is assumed to be a potentially corrupted version of some correct program. The point between two adjacent sentence fragments is referred to as a local error detection point.
(after [13]), and is an indication of at least one error (and possibly more) somewhere in the vicinity of that point. The goal of extended parsing is to locate, within a given input string, the least number of reasonable sentence fragments and, for each fragment, determine its local structure with respect to a particular grammar. Given an interpretation of a string as a particular sequence of fragments, the goal of error reparation is to determine a repair locally around each local error detection point which preserves the maximum amount of the original string's local structure (context-sensitive repair). This yields an interpretation of the original string on which semantic processing may continue.

Note that, in this framework, the error reparation process is independent of the parsing technique and requires only knowledge of the grammar and language for its specification. This permits systematic investigation of (practical) automatic syntax-directed error analysis to proceed independently of a particular parsing technique without necessarily suffering from excessive generality.

The extended parsing method is initially developed in its full generality by automatically augmenting a grammar with a number of error productions which generate all possible valid fragments of nonzero length. This augmented grammar is then used to parse strings with a modified version of Earley's algorithm [14]. The modifications are similar to those used in Peterson's
minimum distance error detection technique [49]. While this very general approach is probably not useful in a practical compiler, it serves as a basis for understanding the ad hoc recovery strategy of Graham and Rhodes [20] and related techniques.

A practical extended parsing technique is then developed from insight gained through analyzing the relationship of the general extended parsing strategy to the work of Graham and Rhodes [20] and Druseikis and Ripley [13]. This technique is demonstrated as an extension of the canonical LR(k) parsing method [34], and results in a parser which can process any input string in linear time, producing an approximation to an extended parse of the string. The parser is an approximation in the sense that, by restricting its operation to deterministic left-to-right processing of the input string, the recognition of fragments is biased to the left. Additionally, the parser will in general contain states in which a nondeterministic choice of the next move is required for certain inputs. In order to allow deterministic processing, configurations which require such choices are treated as if a local error had been detected. This parser is an improvement over the work of Druseikis and Ripley [13] and Pennello and DeRemer [48] in both efficiency and performance. Application of the basic technique to the $\text{SLR}(k)$ and $\text{LALR}(k)$ parsers [11] is immediate through the use of the characteristic $\text{LR}(k)$
parser construction of Geller and Harrison [19]. It is also easily applied to other parsing techniques.

Certain instances of inadequate states can be resolved by using additional lookahead, or by the parseahead technique of Fraley [18]. Alternatively, a simple right-to-left resolution process applicable to the condensed input string after the initial left-to-right pass is described. This technique is related to methods used by Modry [45] and Pennello and DeRemer [48], and may be used to eliminate those spurious local error points which can be resolved by limited right context. Algorithmic and heuristic methods of eliminating inadequate states are also described.
Chapter 2
REVIEW OF RELATED RESEARCH

2.1 Introduction

Previous work in syntax error analysis can be roughly divided into two categories according to their basic strategies: local or global. The local strategies are characterized by their common approach whereby changes are made to a parser's configuration using only the local context available at the error detection point. The global strategies generally approach the problem in terms of minimum distance changes to the input to achieve a globally optimal reparation, resulting in basically nondeterministic solutions. Here, the term "nondeterministic" means a deterministic computation of all possible interpretations followed by a choice from among the best interpretations. This approach allows the utilization of extended context in making a decision.

In this chapter, various approaches to the problem in both categories are briefly presented to give the reader some understanding of the range of possibilities. This selection is by no means complete. Other interesting results not reported here may be found in [4,12,16,26,47].
2.2 Local strategies

The first treatment of error recovery in "automatically generated" parsers is due to Irons [25]. His scheme is based upon a multiple-tracking top-down parser which maintains all possible parse paths in parallel. Errors are detected at the point where none of the parse paths can be continued, which should be at or shortly after the actual error according to Irons. The error recovery routine then builds a list \( L \) of all the symbols (both terminal and non-terminal) in the incomplete branches of the parse, and scans and discards input symbols until a symbol \( A \) is found such that \( X \Rightarrow^* \gamma A \alpha \) for \( \alpha \in \Sigma^* \) and \( X \) in \( L \). The most recently applied production that generated \( X \) is determined, and the string of symbols between the current point in the right side of the selected production and any of its currently incomplete descendents and \( X \) is used to generate a terminal string \( z \). The parser will then be able to accept \( zA^\gamma \) and hopefully continue.

Irons' strategy is completely automatic, and, under the assumption that the input string is almost correct, the recovery should usually not have to purge many input symbols before parsing can resume. Furthermore, repair rather than deletion followed by insertion could be done using the selected production as a template, and the technique itself is applicable to the more efficient top-down parsing strategies. This technique guarantees that the parser will eventually get to the end of the input, but errors may be missed during the deletion phase.
The first widely used "compiler generating" system, XPL [41], utilized a simple recovery strategy now popularly referred to as panic mode. The implementor is required to specify a list of synchronizing symbols of the particular language. When an error is detected, the recovery action is to delete input symbols until one of the synchronizers is found, followed by "backing up" the parser by popping the parse stack until the parser's configuration accepts the symbol. If the stack is emptied while searching for a configuration which accepts the "trustworthy" symbol, more input is purged until a symbol which starts a sentence of the language is found or the input string is completely purged.

Successful application of this recovery strategy requires knowledge of special characteristics of the grammar, the language and the recovery scheme. Attention to the form of productions in the grammar can minimize backup and enhance the chances of finding a configuration which accepts a particular symbol. Variations on this basic panic mode strategy of recovery have been proposed to overcome some of the deficiencies. One simple improvement used in the BLISS/ll compiler [63] is to temporarily halt a purging operation when a symbol is encountered which uniquely defines the start of a major language construct, such as an opening bracket symbol. At this point, the parser is called recursively to parse the internal structure of that construct. When the recursive
invocation returns, the purging operation is resumed. The intent of this enhancement is to avoid the cascaded errors caused by purging such starting symbols.

Wirth [62] describes two ad hoc techniques which he utilized in a simple precedence parser as an alternative to the panic mode approach. The compiler implementor is required to specify a list of insertion symbols and a set of error productions. When a symbol-pair error occurs (no precedence relation holds between the top symbol on the stack and the next input symbol), the insertion list is scanned for a symbol which can occur legally between the erroneous pair. If one is found, it is inserted, otherwise the next input symbol is stacked. When a reduction error occurs (the potential phrase is not a valid right side), the error productions are scanned for an applicable production.

This approach allows the implementor to attach special error diagnostics to those situations handled by error productions, and can achieve good results for those specific forseen situations. Effective utilization requires a thorough understanding of the parser's operation with respect to the particular grammar, and of the probable misuse of the particular language. Handling all errors by this method would require a great number of error productions, so an alternate scheme is required to take care of situations not predefined.
The philosophy behind Wirth's approach is that it should be possible to find very local changes to the input at the error detection point which will make the input locally correct. He contends that most of the committed errors exhibit strong similarities which can be analyzed by a relatively simple algorithm. Since any heuristic scheme for error recovery must make reasonably restricted assumptions about the possible error configurations, the important constraint, according to Wirth, is that the common errors are handled intelligently.

Error productions are an appealing idea, since the implementor has the ability to associate semantic actions with error configurations. The difficulty with Wirth's approach is that the exact structure of each anticipated error must be specified. This disadvantage is eliminated with an extremely simplified notion of an error production that is used in conjunction with panic mode recovery schemes for LR(k) parsers [1,29,50]. With this technique, the compiler implementor has the facilities for specifying semantic actions to be executed at error detection points based on the local context of the error, but is not required to specify the exact context in advance.

Poonen [50] and Aho and Johnson [1] describe techniques in which a special terminal symbol, ERROR, is allowed as the right side of productions in the grammar used to construct some type of LR(k) parser.
Upon encountering an error, Aho and Johnson's technique searches the stack for the topmost state which can shift the ERROR symbol, making it the top of the stack. The ERROR symbol replaces the current input symbol, and can be shifted onto the stack. Various ad hoc approaches can then be used to allow the parse to continue, such as allowing the semantic routine associated with the particular error production recognized to fix up the configuration, or purging the input string to a follower of the left side of the error production. Poonen's method is slightly different, in that the first symbol which can follow the ERROR symbol from any state on the stack is located in the input string, and the corresponding state in the stack is made the top state. The ERROR symbol is then shifted and the next input symbol will force a reduction by the appropriate error production and can eventually be shifted itself.

A somewhat more complicated version of this same basic approach is provided with YACC [29], an LR parser generator available on the UNIX [52] system. An error production may be of the form A \rightarrow \text{ERROR } \alpha, where \alpha is a (possibly empty) string of grammar symbols. Upon shifting ERROR after popping the stack as before, the input string is purged until a symbol which can start \alpha is found and parsing continues normally.
This simplified version of error productions has the advantage that the implementor need be concerned only with the grammatical structure of the language. In effect, he or she is allowed to specify a set of goals for recovery by error isolation through reduction. The recovery process generates in each particular instance of an error the necessary right side, corresponding to the purged stack and input symbols.

The major drawback common to all of the methods reviewed so far is that the implementor is required to specify some aspect of the error recovery process. While there is no objection to allowing this flexibility, it would be more desirable for this to be an optional feature rather than a requirement.

Recovery by reduction is given a more general treatment by Leinius [34] based entirely on the structure of the grammar used to define the language. Leinius contends that, since the phrase is the basic syntactic unit used to specify the logical structure of a context-free language, it is natural for any recovery strategy to depend on the phrase structure of a string. Most other approaches attack the problem at either a lower level (deletion, insertion or replacement of symbols) or at a higher but less structured level, such as the panic mode notion of synchronizing points.
Leinius' method identifies the smallest potential phrase containing the point of error detection which is required by its immediate context to reduce to some unique nonterminal of the grammar. This technique essentially computes the minimum synchronizing point based on the context around the error. The uniquely specified nonterminal replaces the error phrase, and normal processing continues.

There are several drawbacks to Leinius' approach. Performance of the recovery is highly dependent on the form of the grammar, requiring that the implementor tune the recovery by using appropriate productions [27]. This "recovery programming" can be complicated by the restrictions of certain parsing techniques, such as simple precedence which was the parsing technique used by Leinius to develop the technique. He does, however, briefly develop the method for canonical LR(k) parsing. The limited context used in making a replacement decision and the unique replacement criterion degrade performance somewhat, and often lead to situations where several elements of the parse stack must be discarded.

The basic technique of error isolation proposed by Leinius has received a lot of attention and refinement. Sippu and Soisalon-Soininen [55] formalize his recovery method in a generative model, and develop methods of relaxing some of the restrictions inherent in the original approach. Leinius' original canonical LR(k) technique has
been implemented by James [27] and more recently by Boullier [7], and was generalized by Peterson [49] for applicability to reduced versions of LR(k) parsers in addition to the canonical version. Peterson's method was implemented by Carter and Zucconi [8] for the SLR(1) case.

Graham and Rhodes [20] present an error repair technique based on Leinius' idea of error isolation, but use the internal resemblance of the error phrase to some right side of a production in choosing a replacement nonterminal. Their technique utilizes the locality property of precedence languages, and is demonstrated for the simple precedence case.

The property of locality is intuitively the ability to detect the handle of a right sentential form given only the surrounding local context. For simple precedence languages, the only context required is the symbol immediately preceding and the symbol immediately following the handle. The strategy used by Graham and Rhodes involves backward and forward context gathering actions which attempt to detect phrases surrounding the point of error detection. This technique assumes any valid right context at the error detection point for the backward move, allowing all possible reductions to occur on the top of the stack. Similarly, any valid left context is assumed at the error detection point for the forward move, allowing reductions to occur to the right of the error point. This condensed
context surrounding the error point is then used in a pattern matching process to select a reduction goal for the error phrase. The authors note that this strategy is especially good, compared with other methods, at recovering locally from bracketing errors and errors within list constructions. Rhodes [51] observes that his approach appears to discover more of the errors when the errors are dense, and, in general, to give fewer spurious error indications than other methods.

Because of the highly successful results of Graham and Rhodes method compared to previous "automatic" techniques, recent work [13,45,48] has attempted to utilize similar error isolation techniques with the LR(k) parsers and their reduced variants. The consensus is that the backward move technique is not generally applicable due to the correct prefix property of these recognition schemes. Attention has therefore focused on the forward move technique, and has resulted in significant advances in LR(k) recovery strategies.

Modry [45] describes a method which runs all possible parses in parallel from the point of error detection. Each state in the LR(k) finite-state control which can shift the next input symbol begins a possible parse. No fixed forward bound is used, and all forward moves continue until each has attempted to reduce over the error point or has encountered another error. Any parse attempting to reduce over the error point is immediately processed by
the reparation procedure so that it can continue forward (if it can be repaired).

Modry's reparation process attempts to use information contained in the LR states on the left and right of the error point to make a parse continuous across the error point. The method described uses the tokenized input string, considering deletion of symbols and insertion of a single symbol based on insertion costs supplied for each terminal symbol. When all forward moves have either been repaired or terminated (due to encountering another error or not being repairable), one of the repaired parses is chosen as the recovery. If all parses terminated, Modry attempts to repair those which detected a second error at least two symbols past the first error.

Modry's approach demonstrates the advantages of extended right context in making recovery decisions. Errors which are arbitrarily far to the left of the parser-defined error can be discovered when right context indicates that the error point can be moved to the left. The approach as presented suffers from the restriction to a single insertion; in fact, as noted by Modry himself, a fixed number of changes will always fail to repair certain errors. This means that some other recovery technique is needed as an alternative to handle those failures. The primary problem with this approach, however, is the time and space required to implement the parallel forward moves.
Druseikis and Ripley [13] achieved the first (published) practical application of recovery by the forward move technique for SLR(1) parsers. Their method involves constructing an SLR(1) parser augmented with a set of restart states (and all of the successors of these restart states), one for each symbol in the grammar. A restart state and its successors for symbol X is constructed similarly to DeRemer's [11] method for constructing SLR(1) parsers. The restart state itself has a corresponding item set containing LR(0)-items which have the marker to the left of an X. Thus, the restart state for X is essentially the union of those item sets from the set of LR(0) item sets defined by the original grammar which have an X-successor, where items other than X-items (an item with the marker to the left of an X) are mostly irrelevant. Successor sets are computed in the standard way.

The restart item sets and those successors which are disjoint from the LR(0) item sets computed by the standard method from the grammar are then used to construct recovery states, with conflicting actions resolved (in some cases) by the use of DeRemer's technique of adding lookahead. Some recovery states are likely to remain inadequate after lookahead augmentation. This results from the fact that recovery states attempt to recognise suffixes of phrases without benefit of left context, so that some strings may have ambiguous interpretations. When an error is
detected during parsing, the initial recovery action is to start the parser in the restart state corresponding to the illegal symbol. Parsing can then continue forward until another error is detected, an inadequate recovery state is entered, or a reduction is attempted across the error detection point. This last possibility requires that the parser check the stack each time a reduction is called for after an error to determine if the error detection point is included.

Druseikis and Ripley present a simple recovery strategy which always enables the parser to move forward from an error configuration, an inadequate recovery state, or a configuration requiring a reduction across the most recent error detection point. This strategy involves an ad hoc decision as to which restart state to enter in some cases. The approach has the advantages of speed and of preventing many cases of cascading error detection. The authors claim that this technique enables reasonable local diagnosis of those error detectable from the point of the last error (referred to as local errors). Tai [56], however, points out that since no repair is done to the parse stack, error diagnosis performed during the left-to-right parse may tend to appear inconsistent, especially if the errors are close together. It is possible, however, to utilize the right context gained up to the next restart point in heuristic repair of the error configuration, as done by Graham and Rhodes.
Pennello and DeRemer [48] achieve the same effect as the recovery of Druseikis and Ripley, but with a construction technique applied to the finite-state control of the parser, rather than directly within the parser construction algorithm. They compute a forward move stack machine with each state a subset of states from the base parser having the same entering symbol. The actual members of a particular state set are determined by its predecessor state. Rather than have a set of restart states, one for each grammar symbol, a single state is constructed composed of all those states of the base parser which have a successor under any terminal symbol. The initial recovery action is to start the forward move machine in its initial state (the restart state) and continue forward until an error is detected, an inadequate state is entered, or a reduction across the error detection point is attempted.

Although the technique is demonstrated with LR(k) parsers, it appears generally applicable to any parser represented by a finite-state control component. This method is an improvement over that of Druseikis and Ripley because of the ability to utilize the more discriminating resolution techniques involved in the .LR(k) and .LALR(k) parser construction algorithms. The method still requires checking the stack before each reduction after an error is detected to determine whether it is
across the error point. The basic recovery strategy employed is to execute the forward move machine until it halts due to an inadequate state, another error, or a reduction attempted across the error point. The right context gained is then used by heuristic repair strategies.

The approaches used by Druseikis and Ripley or Pennello and DeReemer are seen as attempts to achieve, by simulating parallel interpretations during parser construction time, what Modry does by parallel simulation at parsing time. At a point where Modry follows two different interpretations forward, these deterministic techniques must halt. Thus, Modry is better able to localize the error, but pays for the advantage with loss of efficiency.

McGruther [40] approaches the problem of error isolation by restricting consideration to grammars which are both LR(k) and RL(k). Upon detecting an error, his strategy is to skip forward a "safe" distance in the input string and then look for a reverse synchronizing symbol. These symbols are those which uniquely define some state in the RL(k) parser's finite-state control. The RL(k) parser is then started in the selected state and parses back toward the error detection point. This action halts when an error is detected or a reduction occurs across the starting point. The right context supplied by the reverse parse is then used to assist the error analysis. Unfortunately, this LR/RL requirement appears to be overly restrictive.
Fischer, et.al. [17], have developed a method of error repair for a subclass of \( \mathcal{LL}(l) \) grammars with attractive properties. The technique operates by computing a locally optimal (least cost) insertion string at each error detection point. The insertion-only philosophy of error recovery is based on the observation that, for a grammar in a suitable form, it is feasible to build a correct program around the input string submitted to the compiler. This approach is appealing because the resulting error repairing parser is linear in time and space and produces a syntactically correct program from any input string. Recovery tables used to drive the process are computed from the grammar and a table of symbol insertion costs. This allows the implementor to fine tune the recovery based on experience with the compiler, but unit insertion costs could be assumed for the initial implementation.

There are two objections to this method. The first is that not all \( \mathcal{LL}(l) \) languages are generated by a grammar in the insertion-only repairable class. This can be handled by the standard technique of defining a superset of the desired language and checking the derivation tree in the semantic phase. For example, to handle the case of errors in the block structure of an Algol-like language, which may result in the parser reaching the logical end of a sentence before the physical end of the program, a program can be defined as a sequence of blocks rather than just one.
Effective use of this approach requires a thorough understanding of the possible errors that can occur in the language.

The second major drawback is that an insertion-only interpretation of the input may drastically alter the structure of an otherwise almost correct program. For example, replacement of a colon by a semicolon in the bounds of an array declaration in Algol60 may result in extensive insertions due to limited right context following the error point. In any case, the most reasonable interpretation would be to simply replace the semicolon by a colon. The authors note that this situation could be handled by extending the technique to include deletion, but the speed of the process may well suffer.

Both LaFrance [32] and Tai [56] approach the problem of error repair on the premise that strings presented to the parser will ordinarily differ only slightly from legal strings. Therefore, it should be possible to find very local changes to the input string at the point of error detection which make the input locally correct. These schemes use pattern mapping techniques which attempt to choose one of a set of predetermined patterns which will transform the substring in the vicinity of the error into a locally correct substring.
LaFrance's method is ad hoc, and Tai essentially supplies the formal basis for LaFrance's approach. A pattern here is a mapping specifying how to rearrange (including ignore) the symbols found locally around the error point and where to insert new symbols. Unfortunately, it is impossible to repair all types of syntax errors with this technique due to its very local nature. Provision is therefore required for some other recovery strategy in the event that this one fails. Boullier [7] uses this method in combination with Leinius' technique for LR(k) parsers.

Meertens and Van Vliet [44] describe a top-down error recovery method for a subclass of the operator-precedence languages. The method relies on an initial left-to-right pass which obtains a globally optimal correction of the bracketing structure of the program [42, 43], followed by a right-to-left pass transducing the (partially) repaired input into prefix form, which is then parsed by an LL(1) parser. This method is oriented toward handling the complexity of the parenthesis structure of Algol68 [61]. The goal of this complicated processing is to increase the points in the input string where parser resynchronization can take place after an error. Because global bracketing errors are handled by the initial pass, local errors can be specified by error productions added to the prefix-form grammar.
The prefix-form transducer is constructed automatically, under the assumption of correct bracketing structure, for a subclass of operator-precedence grammars referred to as operator-parenthesis grammars. The essence of this class is a refinement of the notion of operator into mutually disjoint sets of symbols characterizing the general types of parenthesis symbols of Algo168: [61].
2.3 Global strategies

Levy [35, 36] develops a formal model for optimal local repair of syntax errors with a left-to-right parsing strategy. His approach is based on the premise that an incorrect input string consists of clusters of errors separated by enough locally correct input to be distinguishable. This allows the use of an error-repairing mode of parsing only when errors are detected, proceeding from left to right treating each cluster independently.

The basic objective in this model is to bound the area where the error mode is required. Levy approaches this by determining the least number of input symbols to the left of a parser-defined error in which a repair may be needed and before which no repair is needed. Following this backward move, a forward move constructs all possible local interpretations of the input with a fixed maximum number of edit operations, starting from the position backed up to. The finite-state control of the base parser is used to derive the forward parses. This forward move phase continues until all interpretations are equivalent in the sense that any one of them may be selected without interfering with the interpretation of the next error cluster, or until all interpretations reach the fixed edit bound without becoming equivalent.
The restriction to a fixed maximum number of local minimum distance edit errors per error cluster implies that an interval of the input string which is too dense with errors cannot be handled. Furthermore, it turns out that there are no reasonable bounds on the backward and forward move distance without additional heuristic assumptions about the language and types of errors. This approach is used successfully in an interactive incremental compiler by Tindall [58].

Peterson [49] develops a minimum distance error repairing parser based on Earley's algorithm [14, 15] which maintains the \( O(n^3) \) time and \( O(n^2) \) space bounds established by Earley in [14] for a string of length \( n \). The method involves extending the original grammar defining the language to be parsed with a set of error productions so that the resulting grammar generates any string in \( \Sigma^* \). Basically, each terminal symbol in the original grammar is replaced by a regular set describing how all possible edit errors may occur to that terminal symbol in a sentence to generate an incorrect string. Earley's algorithm is extended slightly to keep track of the number of edit errors assigned to each parse of the input string, with the goal of choosing a parse containing as few error productions as possible. The particular error productions used in a selected parse indicate the positions and types of minimum distance errors in the string a posteriori.
An objection to this particular approach is that the extended grammar is extremely ambiguous, so that the constants of proportionality in the time and space bounds are likely to be extremely degraded with respect to the parse of a correct string using Earley's technique and the original grammar. Furthermore, since potential interpretations of an incorrect string are generated at the token level using no context or knowledge about the phrase structure of the particular language, there is no guarantee that a particular minimum distance interpretation will bear any similarity to the original input string.

Lyon [38] also develops a minimum distance error repair algorithm based on Earley's parser. His approach is to build an error interpretation mechanism directly into the parsing algorithm rather than in the form of error productions. This allows him to use Bellman's dynamic programming technique [5] to reduce somewhat the number of interpretations assigned to any substring, thus saving some space over Peterson's approach. The time and space bounds are the same, however.

Teitelbaum [57] approaches the minimum distance error analysis problem in a very general way. He introduces the notion of weighted grammars, which can define functions from $\Sigma^*$ into some commutative semiring $R$, called algebraic power series in non-commuting variables. These power series can be evaluated in $O(n^3)$ time and can be used to define
minimum distance error measures between a given language and $\Sigma^*$. The minimal number of errors in an arbitrary string is a algebraic power series.
2.4 Review

The basic approach of most of the local strategies discussed in this chapter is to externally manipulate a parser's configuration after it detects an error (and halts) in order to enable the parser to continue forward and detect further errors. Modifications are accomplished either through immediate repair of the error based on the available context, or through recovery, by some form of backup followed by skipping forward until a non-error configuration past the error detection point is found. Both techniques suffer from dependency on the parsing algorithm. The alternative to recovery and repair is to use error productions, so that the grammar generates all possible error situations and the parser need never halt. Since this is often difficult to do by hand, the usual approach is a limited version of this in combination with some other strategy for recovery.

The global strategies, on the other hand, essentially construct all possible interpretations of the input string and then pick the optimum interpretation under some heuristic assumptions of reasonableness. The difficulty here is the time and space required to perform this computation on realistically sized programs. Many of these strategies also take the same time and space regardless of whether the string is correct or not.
Neither the local strategies nor the global strategies are completely satisfactory in themselves, which has led some researchers to combine aspects of the two strategies (see [35,36,44]). Such combinations do not necessarily avoid the drawbacks to either approach, but this type of solution seems to offer the best chances of reasonable performance in both error analysis and efficiency.
Chapter 3
ERROR DETECTION WITHOUT RECOVERY

3.1 Motivation

The ultimate objective of syntax error analysis in a compiler is to identify (and diagnose) the least number of reasonable syntax errors in a program. Unfortunately, there is little agreement on what constitutes a reasonable interpretation of errors in some arbitrary incorrect program. The formal models of syntax errors found in the literature [2,36,38] are based on the minimum edit distance model. Rhodes [51] contends, however, that minimum edit distance can be a poor basis for reasonableness because of its sensitivity to the particular distance metric used, and the fact that there is no guarantee that the minimum distance error interpretation bears any similarity to the intended program.

The so-called practical results in automatic syntax error analysis have taken the approach of binding the process of repair (and diagnosis) directly to the parser's detection of errors. This is not an intrinsic constraint of syntax error analysis, but of the method of implementation. The approach widely adopted for constructing an error repairing (or recovering) analyzer is to augment a particular parsing algorithm with an external algorithm performing the error analysis. The standard parsing
algorithms are all based on the sentence recognition process, so that a parser halts when the existence of an error is detected. In order to permit detection of additional errors, the error analysis algorithm is required to perform some kind of recovery action on the parser's configuration so that parsing can continue. Thus, the error analysis tends to be in part directly based on the particular parsing technique.

The capabilities of a syntax error analyzer are clearly hampered by the restricted right context and left-to-right bias inherent in this approach. The details of previous repairs are difficult to reconstruct from the reduced contents of the stack. This makes it difficult to reevaluate responses to prior errors, so typical repair strategies simply treat each new parser-defined error independently. This results in a tendency to detect spurious errors. Some of the more recent methods [13,21, 48] have demonstrated the potential for superior local performance due to various techniques for extending the right context available at a particular error detection point. These methods still suffer from a left-to-right bias, which may result in some cascading effects. However, massive local cascading is largely eliminated due to the extended right context utilized in evaluating a repair.
There appears to be no a priori requirement that error detection and reparation operate in parallel in order to achieve efficient solutions. The justification offered is that it makes use of efficient parsing algorithms which are already available. The only intrinsic constraint on syntax error analysis is that the detection and location of errors is a prerequisite to the repair of those errors. The location of a parser defined error is only an indication that at least one programmer error occurs somewhere in the vicinity. Hence, this "bolt-on" approach to error detection is no more than an approximation to an optimal solution to the problem of automatic syntax error analysis. Some other approximation technique may yield a better solution. An alternate approach to the implementation of automatic syntax error analysis is therefore proposed.

Bellman's principle of optimality, on which dynamic programming is based, states

"If an objective function is separable into independent segments, then each segment may be individually optimized and the results combined into a final optimum solution." [5]

This principle suggests a serial decomposition of syntax error analysis into error detection followed by error reparation and diagnosis. This reflects the natural dependency of the two subprocesses. In the parsing process, the canonical parse of a sentence is (and should be) independent of the parsing algorithm. Likewise, the detection of an error should be independent of any repair
actions made in response to other errors in the string, and the repair of a particular error should be independent of the particular detection algorithm. The only real constraint is that repair of an error depends on the location of the error in a string - the particular context of the error.

Several advantages to this approach are immediately apparent. Since the error diagnosis and reparation phase is totally independent of the error detection technique, it can be designed using heuristics based solely on the language and its grammar. This at least allows the problem to be investigated independently of any parsing technique, and may possibly lead to the development of truly automatic generation of this phase of a compiler. With no unnecessary interdependencies, the sophistication of the error detection and the error reparation phases may be increased without interfering with each other. As an added by product, the working set size of the syntax analysis phase of the compiler is potentially reduced for incorrect programs. Finally, the global context of each error may be considered in deciding on a diagnosis and repair, using such knowledge as the density and extent of errors, plus knowledge made available from any semantic analysis that can be performed on the incorrect string.
A potential drawback to this approach is that there is no adequate model of syntax error detection for context-free languages. Existing sentence recognition algorithms only detect the existence of the first error in a string and require some kind of recovery algorithm to continue through the string. Sentence recognition, as currently formulated, is an inadequate model of the error detection process in a practical compiler without this basically ad hoc extension. Error recovery, however it may be actually accomplished, approaches the problem of error by externally manipulating a parser's configuration. A preferrable approach would seem to be to modify the model of sentence recognition on which parsing is based to encompass detection of errors rather than simply detection of nonmembership in some language. This requires a model of incorrect strings.

Levy [35] states that, pragmatically, it is not necessary to repair programs which contain too many errors, since the cause for a great number of errors is likely to require intelligent attention for diagnosis. He interprets the phrase "too many" in this context to mean "locally too dense". Under the premise that the program submitted to a compiler was intended to be correct, Levy assumes that errors occur in clusters which are far enough apart to be easily distinguishable. Using this model of an incorrect program, error repair is required only in the vicinity of each error cluster with context around the cluster
available to evaluate the repair. Furthermore, there is some assurance that repairs made to different clusters will not interfere with one another by requiring that clusters be separated by enough correct input to uniquely determine the optimal equivalence class of reparations to the cluster.

The error cluster model treats incorrect programs as segments of locally correct program pieces separated by clusters of errors. By assumption, a locally correct program piece is longer than the interval encompassed by the error cluster on either side, since otherwise the two clusters could not be distinguished. This is, intuitively, a natural interpretation of the notion of an incorrect program. However, the errors in a program, in Levy's model, are defined a posteriori by the repairs made in the vicinity of the parser-defined error. This is necessary because Levy considers deletion, insertion and replacement errors with the goal of finding a least cost reparation, and so must construct all possible interpretations and then select the best one.

The usual model of sentence recognition is given in terms of sentential forms defined by a particular grammar generating the language. The phrase, in this generative model, is the basic syntactic unit used in specifying the structure of the language, and it is this structure on which the recognition process is based. The general idea of the procedure is to determine a phrase in some potential
sentential form (initially the input string), and then replace it by the nonterminal which generates that phrase according to the grammar. Leinius [34] contends that it is natural for syntax error recovery to depend on the phrase structure of an incorrect string. Likewise, it seems natural to determine the locations of syntax errors based on the incorrect phrase structure of a string. This contrasts with the (less structured) minimum edit distance techniques, which essentially approach the problem at the interface between the lexical and syntactic structure of the language.
3.2 The error detection model

From the point of view of an algorithm which intends to detect errors before reparation is attempted, deletion errors are the easiest to detect. This is because every symbol in the input string is, a priori, an equally likely candidate for a replacement or insertion error. On the other hand, by assuming that only deletion errors occur, those points in the input string where symbols must be inserted to make the program correct will not generally be as numerous under the assumption of nearly-correct inputs. This is similar to Levy's model, where now an incorrect program is treated as a sequence of locally correct program pieces separated by deletion error points. Furthermore, the incorrect phrase structure of a string can be used to determine the location of syntax errors in this case.

Before proceeding, it should be noted that a "deletion-errors only" interpretation of an incorrect program is not generally a reasonable interpretation, as mentioned in Chapter 2 in the discussion of the insertion-only repair strategy of Fischer, et. al. [17]. However, by divorcing reparation from error detection, the repair algorithm need not adhere to such an interpretation. Instead, the interpretation determined by the error detection phase should be viewed as the best approximation that can be made to the structure of the string without additional information if the string contains errors.
This guess may then be reevaluated by the reparation phase based on global knowledge about the incorrect structure.

The remainder of this section develops a model of sentence recognition which includes error detection as an integral concept. This model will then be used to develop techniques of "parsing" an arbitrary string in $\Sigma^*$ in the last section of this chapter and in Chapter 4. Unless otherwise specified, the definitions in this section assume a context-free grammar $G = (N, \Sigma, P, Z)$.

**Definition 3.1**

A string $\gamma \in \Sigma^*$ is a valid fragment [48] of $G$ if there exists $\alpha, \beta \in \Sigma^*$ such that $\alpha$ is a prefix of some sentential form of $G$ and $\alpha = \beta \gamma$.

**Definition 3.2**

Let $\alpha \in \Sigma^*$ be a valid fragment of $G$. For each $x \in \Sigma^*$ such that $\alpha \Rightarrow^* x$, $x$ is a locally correct string.

Notice that a valid prefix is a valid fragment, and a sentence is locally correct. A locally correct string is also a valid fragment.
Lemma 3.1
Let $G = (N, S, P, Z)$ be a context-free grammar and $x \in \Sigma^*$. If $x$ is a locally correct string then $x$ is a valid fragment of $G$.

Proof
Immediate.

Definition 3.3
Let $z \in \Sigma^*$ and $z = u x w y v$ for some $u, v, w, x, y \in \Sigma^*$. If $x w, w y$ are both locally correct strings but $x w y$ is not a locally correct string, then one (local) error with respect to $G$ is said to exist between $x$ and $y$ in $z$. The local error is contained in $w$, and $w$ is a (local) error interval with respect to $G$ of $z$

The concepts introduced in Definition 3.3 are illustrated in Example 3.1.

A typical parsing algorithm is capable of detecting a single local error in a string, i.e., the parser-defined error in the string. This is the first local error detectable in a single left-to-right scan of the input by the particular parser. However, these parsing techniques are not generally capable of determining local error intervals containing the parser defined error, nor can they detect more than one local error. Note that if an
Example 3.1

Consider the following string of symbols with respect to a context-free description of the syntax of PASCAL [28]:

\[
\text{IF } A + B ) \text{ THEN}
\]

It should be apparent that this is not a locally correct string. There are four possibilities for the position of a local error within this string, as indicated, but there is only one local error. This string can be split into locally correct segments in various ways, as indicated below, and the longest local error interval is w.

\[
\text{IF } A + B ) \text{ THEN}
\]

\[
| \quad | \quad | \quad | \quad | \quad | w \quad | \quad |
\]

\[
| \quad | \quad | \quad | \quad | \quad | \quad |
\]

\[
| \quad | \quad | \quad | \quad | \quad |
\]

\[
| \quad | \quad | \quad | \quad |
\]
error interval has non-zero length, then there maybe more than one equally plausible local error point contained within the interval.

**Lemma 3.2**

Let $G = (N, \Sigma, P, Z)$ be a context-free grammar, and $z \in \Sigma^*$ such that $z = uxwyv$ for some $u, v, w, x, y \in \Sigma^*$. If $w$ is a local error interval of $z$ with $xw, wy$ both locally correct strings, then $z \notin L(G)$.

**Proof**

Assume $w$ is the leftmost error interval of $z$. This may be done without loss of generality since if there is more than one, there must be a leftmost one. Furthermore, suppose $z \in L(G)$, so that $Z \xrightarrow{\ast} z$. Now, since $Z \xrightarrow{\ast} uxwyv$, $uxwy$ is a correct prefix of $G$ and therefore $xwy$ is a valid fragment such that $xwy \xrightarrow{\ast} xwy$. Hence $xwy$ is a locally correct string, contradicting the assumption that $w$ is a local error interval.

//

**Definition 3.4**

Let $z \in \Sigma^*$ such that $z = uxwyv$ for some $u, v, w, x, y \in \Sigma^*$ with $w$ a local error interval of $z$. The **location** of the local error contained within $w$ in $z$ is denoted by an integer $i$ such that

$$0 \leq \lg(ux) \leq i \leq \lg(uxw) \leq \lg(z).$$
The local error interval \( w \) in \( z \) is denoted by the pair of integers \((i, j)\) such that

\[
0 \leq \lg(ux) \leq i \leq j \leq \lg(uxw) \leq \lg(z).
\]

Two local errors are **different** if their locations are not in the same local error interval, otherwise they are **equivalent**.

This definition allows the specification of the location of local errors in a string either as particular positions or as intervals within the string. The use of error intervals may prove useful in some cases during heuristic error reparation, since the width of a particular interval is likely a measure of the redundancy of the phrase structure in the vicinity of the interval. The width of an interval also depends on the proximity of other local errors, and so is some measure of the density of local errors in the vicinity. Unfortunately, there is no a priori bound on the width of an error interval, and in certain cases an error interval may encompass an entire input string (see Example 3.2). This is related to the problems with the backward move proposed by Levy [35] being unbounded. Henceforth, local errors, by which is meant local error intervals of length zero, will be used rather than local error intervals to avoid this problem.
Example 3.2

The following Algol-like program is made incorrect by deleting the indicated BEGIN, with the longest error interval consisting of the entire string. From the point of view of a parser, the error can be repaired by inserting a BEGIN at any one of the four indicated positions; other reparations are possible, of course.

```
BEGIN INTEGER I, N; READ ( N );
+ + + + +
BEGIN
FOR I := 0 STEP 1 UNTIL N DO PRINT ( I )
END;
I := N
END
```
Definition 3.5
Let $x \in \Sigma^*$ such that $x = x_0 \, x_1 \, \ldots \, x_k$ for some $x_i \in \Sigma^+$, $0 \leq i \leq k \leq \log(x) = n$. If each $x_i$ is a locally correct string for $0 \leq i \leq k$, then $x_0 x_1 \ldots x_k$ is an interpretation of $x$ with respect to $L(G)$. A particular interpretation is denoted by the sequence of integers

$$\sigma = 0, i_1, i_2, \ldots, i_k, n$$

where $i_j = \log(x_0 \ldots x_{j-1})$ for $1 \leq j \leq k$ and $n = \log(x)$,

$$0 < i_1 < i_2 < \ldots < i_k < n.$$

The integers in an interpretation indicate positions in the string between adjacent symbols, or at the end points. Such positions may be the locations of local errors if the substrings on either side cannot be concatenated to form a locally correct string. Notice that Definition 3.5 does not admit interpretations of the empty string. This will be remedied later.

Definition 3.6
Let $x \in \Sigma^*$ and $\sigma$ be some interpretation of $x$. If each $\sigma[i]$, $1 < i < \log(\sigma)$, is the location of a different local error of $x$, then $\sigma$ is a valid interpretation of $x$. If $\log(\sigma) = \lambda$, then the number of local errors in $x$ with respect to $\sigma$ is $\lambda - 2$. 

Lemma 3.3

Let $G = (N, \Sigma, P, Z)$ be a context-free grammar and $x \in L(G)$. If $\sigma$ is a valid interpretation of $x$, then $\sigma = o,n$.

Proof

Since $x \in L(G)$, $x$ contains no local errors.

//

A valid interpretation of some string $x \in \Sigma^*$ under a given grammar is not necessarily unique unless the local error count is zero. This is because an error interval of the form $(i,j)$, where $i < j$, may contain more than one equally reasonable location for the local error. This is demonstrated in Example 3.3. Furthermore, as shown in Example 3.4, the local error count of a string is not necessarily unique.

The local error count as defined does not include local errors which occur at the ends of a string, since valid interpretations do not distinguish between strings with local errors at the end points and strings which have prefixes or suffixes which are prefixes or suffixes of a sentence. This is easily remedied by adding unique end markers.
Example 3.3

The string of Example 3.1, shown here with the positions between symbols numbered, has four valid interpretations. The local error interval containing these local errors is \((1, 4)\).

\[
0 \text{ IF } 1_{2} + 3_{4} \text{ THEN } 5_{6}
\]

\[
\sigma_{0} = 0, 1, 6
\]

\[
\sigma_{1} = 0, 2, 6
\]

\[
\sigma_{2} = 0, 3, 6
\]

\[
\sigma_{3} = 0, 4, 6
\]
Example 3.4

Assuming a context-free grammar for Algol60 with the stipulation that labels may not occur following a THEN and that the assignation symbol is represented by the sequence of symbols ':' followed by '=', the string shown below has two different valid interpretations with different local error counts.

\[
\text{IF}\ A\ \text{THEN}\ L:\ M\ \text{ELSE}\ \text{ELSE}
\]

\[
\sigma_1 = 0, 4, 7
\]

\[
\sigma_2 = 0, 3, 6, 7
\]
Definition 3.7

Let \( G' = (N', \Sigma', P', Z') \) be a context-free grammar. Then the **augmented grammar** \( G = (N, \Sigma, P, Z) \) is defined by

\[
P = P' \cup \{ (Z', |-, Z |-) \}
\]

\[
N = N' \cup \{ Z' \}
\]

\[
\Sigma = \Sigma' \cup \{ |-, -| \}
\]

for some \( Z', |-, -| \not\in V' \).

The symbols \(-\) and \(-\) are the **right** and **left** endmarkers, respectively.

For any context-free grammar, the augmented grammar is obviously also context-free. By assuming that all strings considered have right and left endmarkers only on the left and right ends, respectively, local errors will never occur at the ends of strings.

The notion of valid interpretation is the basis for the model of recognition with error detection used in this dissertation. This model is embodied in the following definitions.

Definition 3.8

A **recognition algorithm** for \( \Sigma^* \) (with respect to \( L(G) \)) is an algorithm which computes a valid interpretation of some string \( y = |-, x |, -| \) where \( x \in \Sigma^* \) according to the augmented grammar.
The local error count of a valid interpretation of some string is not necessarily unique. Furthermore, a particular local error is at best only an indication that at least one programmer error occurs somewhere in the vicinity. Therefore, any valid interpretation supplies about the same amount of information. Minimizing the local error count, then, seems to be a desirable goal, since a valid interpretation with minimal local errors would then be the most compact representation of the errors in a string. In addition, as shown in Example 3.4, the minimal interpretation is more reasonable in some cases and tends to localize the problem better.

Definition 3.9
An error detection algorithm for \( L^* \) (with respect to \( L(G) \)) is a recognition algorithm which determines a valid interpretation with minimal local error count.

Definition 3.9 says nothing about how to actually decide that a particular substring is locally correct, which is obviously a prerequisite to computing a valid interpretation. Notice that if a substring is locally correct it is a valid fragment, and if it is long enough, it should contain phrases of the language which can be reduced by a sequence of productions from the grammar.
Thus, it should be possible to recognize locally correct substrings by a process similar to parsing a sentence.

**Definition 3.10**
Let \( x \in L^* \) be a locally correct string. If there exists a valid fragment \( \alpha \in V^* \) such that \( \alpha \Rightarrow^* x \) and there does not exist another valid fragment \( \beta \in V^* \) with \( \beta \Rightarrow^* x \) such that \( \beta \Rightarrow^+ \alpha \), then \( \alpha \) is a **maximal valid fragment** corresponding to \( x \).

Depending on the amount of redundancy in the language and the length and composition of a locally correct string \( x \), there may be more than one maximal valid fragment corresponding to \( x \). For the purposes of error detection as defined here, any maximal valid fragment will do.

**Definition 3.11**
Let \( x \in \Sigma^* \) be a locally correct string. A **local parse** of \( x \) is the reversal of a reduction sequence, \( \sigma \), which reduces \( x \) to a maximal valid fragment \( \alpha \), so that
\[
\alpha \Rightarrow^\sigma \rightarrow^R x.
\]
A local parse of \( x \) determines a **local derivation** from some maximal valid fragment.

If the grammar is unambiguous, a particular maximal valid fragment has a unique local derivation sequence for some particular locally correct string corresponding to it.
Definition 3.12
Let $G' = (N', \Sigma', P', Z')$ be a context-free grammar and $G$ be the augmented grammar corresponding to $G'$. For some string $y \in \Sigma^*$ such that $y = | - x - |$ for $x \in (\Sigma')^*$, an extended parse of $x$ (with respect to $G$) is a sequence of local parses together with a sequence of maximal valid fragments corresponding to a valid interpretation of $y$.

In order to make use of the notion of "extended parse" in a practical compiler, it should be the case that an extended parse of a sentence corresponds directly to a parse of that sentence.

Lemma 3.4
Let $G'=(N', \Sigma', P', Z')$ be a CFG and $G$ be the augmented grammar. If $x \in L(G')$ then an extended parse of $y = | - x - | \in L(G)$ is a parse of $y$.

Proof
Since $y \in L(G)$, the only valid interpretation of $y$ is $\sigma = o, \lg(y)$. Since the only production of $P$ containing endmarkers is $Z' \rightarrow | - Z - |$, $Z'$ is the only maximal valid fragment. Any local parse from $y$ to $Z'$ must then be a parse of $y$ according to $G'$.
//
Definition 3.13

An extended parser is an error detection algorithm which also computes an extended parse of the input string.

The general model of extended parsing is defined in such a way that the extended parser is able to cope with any arbitrary string $x \in \Sigma^*$. If the string $|\prec - x - |$ happens to be a sentence of the language augmented with endmarkers then an extended parse is simply a parse of $|\prec - x - |$ with respect to the augmented grammar. The parse of $x$ can then be determined trivially. Otherwise, an extended parse is a sequence of local parses of the locally correct substrings of $|\prec - x - |$ determined by the fewest local errors.

The question that immediately arises is whether or not an algorithm for extended parsing exists. The existence of an algorithm is demonstrated in the next section by modifying Earley's parsing algorithm [14].
3.3 Extended parsing

A generative system for describing a string as a sequence of locally correct substrings is easily derived from the grammar used to define a language. Any such sequence has a corresponding interpretation according to the model presented in the previous section, but an interpretation defined in this way is not necessarily valid, since a sentence can be arbitrarily divided into substrings, all of which will always be locally correct. However, by taking the dynamic programming approach [5] of generating all possible interpretations first, the valid interpretations can easily be determined. Given a valid interpretation, computation of the corresponding extended parse is straightforward. This is the approach taken in the current section.

The original grammar is initially augmented with a new production which places endmarkers around the original sentence prototype. An extended grammar is then constructed from the augmented grammar by the addition of error productions. The error productions describe all possible ways to generate locally correct strings which are not phrases of the augmented grammar. The extended grammar is also context-free, and phrases of this grammar can be either phrases or phrase fragments of the augmented grammar. The left sides of the error productions are new symbols not in the original vocabulary so that phrase fragments can be distinguished from phrases of the original grammar in a derivation according to the extended grammar.
Error productions which generate a sequence of phrases and phrase fragments are also added, with the restrictions that a left or right endmarker can only be generated at the left or right end of a string, respectively, and furthermore are always generated in these positions. The language generated by this extended grammar contains any string from $\Sigma^* (\Sigma$ of the original grammar) with unique endmarkers. Therefore, a parse according to the extended grammar of any string with endmarkers will reduce the string to the prototype of the extended grammar, and determines an interpretation of the string.

Earley's parsing algorithm [14] will compute all parses of a sentence according to an arbitrary context-free grammar, even if it is ambiguous. The extended grammar just described is obviously ambiguous, since a sentence of the original language (with endmarkers added) of length $n$ can be generated by a derivation with one valid fragment or with $n$ valid fragments. Therefore, Earley's algorithm, or some other general parsing algorithm, is required. Modifications to Earley's technique allow a parse to be chosen which determines a valid interpretation with minimal local errors.
If a string is a sentence of the original grammar (with endmarkers added), there is at least one parse which determines a valid interpretation with no errors, and there is exactly one such parse if the original grammar is unambiguous. Any parse selected can be used to construct a canonical parse of the string.

If a string is locally correct with respect to some grammar G, then it must be some pattern of symbols corresponding to a phrase, a phrase prefix, a phrase suffix, or a phrase infix of G. This is due to the phrase structure of the context-free languages. Any other pattern of symbols could not be part of a sentence in some phrase structured language, and hence would not be locally correct. By constructing the set of productions derived from the augmented grammar which generate strings according to these four types of error patterns, any locally correct string can be generated.

A production defines a set of prefix error productions by successively truncating a symbol from the right end of the right side. These rules define the ways that the left side directly generates a phrase prefix. In addition, each error production so defined, as well as the original production, whose right side ends in a nonterminal may indirectly generate phrase prefixes by allowing that final nonterminal to generate each of its own phrase prefixes. This same basic process (truncating symbols from one end or both) results in suffix and infix error productions.
Restricting applicability of an error production to the left or right end of an error production is accomplished by the introduction of new nonterminal symbols to the vocabulary, derived from the original nonterminal alphabet. These new symbols replace the left side symbols of all error productions, and indicate whether the right side is a prefix, suffix, or infix phrase. These new symbols are then used in the right sides of error productions in the obvious way to generate phrase fragments of the original grammar in all possible ways.

Straightforward generation of error productions results in some productions which do no useful work, serving only to increase the ambiguity of the extended grammar. In the construction algorithm, these productions are either not generated, or else are eliminated after construction. Error productions with empty right sides are not useful in constructing an interpretation; error productions which define unit cycles \((A \rightarrow A)\) are of no importance; infix single error productions with a nonterminal on the right side are of no importance, since the rules generating a sequence will generate the right side as well as the left side, and infix phrases can occur nowhere else. In order to simplify the exposition, only proper context-free grammars are considered. The construction of an extended grammar is given by Algorithm 3.1. The various primed versions of nonterminals from the original grammar generate fragments of the original grammar.
Algorithm 3.1   Extending a grammar.

Input.  A proper CFG $G = (N', \Sigma', P', S)$.

Output. The extended grammar $G' = (N, \Sigma, P, Z)$ which
        generates sequences of locally correct strings of
        $G$, and the integers $p$ and $p'$. Productions
        numbered 1 to $p$ will be from $P'$; productions
        numbered starting at $p'$ are special error
        productions which will be used during parsing.

Method. Let the productions in $P'$ be numbered 1 to $p$.
        Let $P_e$, the set of error productions, be initially
        empty.

(1) For each $(A, a) \in P'$ with $1g(a) = n > 0$, set

\[ P_e = P_e \cup \{(A, a[i:n]), 1 < i \leq n\} \]

\[ \cup \{(A', a[1:i]), 1 \leq i < n\} \]

\[ \cup \{(A', a[i:j]), 1 < i \leq j < n\}. \]

The primed nonterminals will ultimately generate prefix,
        suffix or infix phrases; 'A indicates a proper suffix
        phrase is generated, $A'$ indicates a proper prefix phrase
        is generated and 'A' indicates a proper infix phrase is
        generated. The prime symbol indicates which end of the
        phrase is truncated.

(2) Eliminate from $P_e$ all single productions of the form
        $(A', A)$, ('A,A) or ('A',A'), and all productions of the
        form $(A', a)$, ('A,a) or ('A',a) such that $(A,a) \in P'$.
        These error productions will only increase the ambiguity
        of a parse.
(3) For each \((A',a) \in P_e\) such that \(a = \beta B\) for some \(\beta \in (V')^*\), \(B \in N'\), add \((A',\beta B')\) to \(P_e\). This allows prefixes of phrases to be generated by a nonterminal ending the right side of an error production.

(4) For each \((A,a) \in P_e\) such that \(a = B\beta\) for some \(\beta \in (V')^*\), \(B \in N'\), add \((A',B\beta)\) to \(P_e\).

(5) For each \((A',a) \in P_e\) such that \(a = \beta B\) for some \(\beta \in (V')^*\), \(B \in N'\), add \((A',\beta B')\) to \(P_e\).

(6) For each \((A',a) \in P_e\) such that \(a = B\beta\) for some \(\beta \in (V')^*\), \(B \in N'\), add \((A',B\beta)\) to \(P_e\).

(7) For each \((A,a) \in P',\) some \(\beta, \gamma \in (V')^*\), \(B, C \in N'\)

(a) if \(a = B\beta\), add \((A',B\beta)\) to \(P_e\),

(b) if \(a = \beta B\), add \((A',\beta B')\) to \(P_e\),

(c) if \(a = B\gamma C\), add \((A',B\gamma C')\) to \(P_e\).

This allows productions from the original grammar to indirectly define phrase fragments.

(8) Let the productions of \(P_e\) be numbered 1 to \(\lambda\).

For some \(\mid-\mid \notin \Sigma', Z \in N'\), add \((Z,\mid - S \mid)\) to \(P_e\) as production number \(\lambda + 1\).

(9) For some \(L,F,R \notin (N' \cup \{ 'A,A', 'A' \mid A \in N' \})\), add the following productions to \(P_e\), in order, numbered starting at \(\lambda + 2:\)

\((Z,LR), (R,FR), (L,\mid -), (L,\mid - S')\), \((L,\mid - S), (R,\mid -)\),

\((R,\mid S -)\), \((R,S \mid -)\)

\((F,D) | D \in (N' \cup \{ 'A,A', 'A' \mid A \in N' \})\), \((D,\epsilon) \notin P'\)
(10) Let \( G' = ( N, \Sigma, P, Z ) \) be the proper CFG defined by

\[
N \subseteq N' \cup \{ 'A', 'A', 'A' \mid A \in N' \} \cup \{ Z, L, F, R \}
\]

\[
\Sigma \subseteq \Sigma' \cup \{ |-, |- | \}
\]

\[
P \subseteq P' \cup P_e
\]

such that the productions of \( P' \) remain numbered 1 to \( p \),

and all productions of \( P_e \) are numbered starting at \( p+1 \)

ordered as for \( P_e \).

Set \( p' \) to \( p+1+4 \); this is the number of production

\( (L, |-) \) in \( P \).
Example 3.5

The construction of an extended grammar by Algorithm 3.1 is demonstrated for the grammar $G = (N', \Sigma', P', S)$ of [49] defined by the following productions:

1. $S \rightarrow AS$
2. $S \rightarrow b$
3. $A \rightarrow Aa$
4. $A \rightarrow b$

The set of productions for $G' = (N, \Sigma, P, Z)$ is shown below, where $p = 4$ and $p' = 14$.

1. $S \rightarrow AS$  
2. $S \rightarrow b$  
3. $A \rightarrow Aa$  
4. $A \rightarrow b$  
5. 'S' 'AS'  
6. 'S' 'AS'  
7. 'S' 'AS'  
8. 'S' 'AS'  
9. 'A' 'Aa'  
10. 'A' 'a'  
11. 'Z' 'S'  
12. 'Z' 'LR'  
13. 'R' 'FR'  
14. L → |-
15. L → | - S'
16. L → | - S
17. R → -|
18. R → 'S' -|
19. R → S -|
20. F + S
21. F + S'
22. F → 'S
23. F → 'S'
24. F → A
25. F → 'A
It is self-evident that Algorithm 3.1 constructs a grammar which generates sequences of locally correct strings enclosed by a pair of endmarkers, and that any string over $\Sigma'$ with endmarkers can be generated. The productions of the new grammar numbered $p'$ and above are the first step in generating a new fragment in a sequence, and have special significance during the parsing process, since these productions are the last reduction made to a particular fragment.

A modified version of Earley's parsing algorithm which counts the number of valid fragments recognized by a particular parse is presented next. The modifications are similar to those used by Peterson [49] in his minimum distance error detection algorithm, and as with Peterson's modifications, do not affect the $O(n^3)$ time and $O(n^2)$ space bounds established by Earley [14] for a string of length $n$. A parse with the least number of valid fragments can then be used to construct the corresponding extended parse of the string in a straightforward manner, provided such a parse corresponds to a valid interpretation, which will be proved.

Given an extended grammar $G' = (N, \Sigma, P, \epsilon)$ constructed by Algorithm 3.1 and an input string $w = a_1 a_2 \ldots a_n$ in $\Sigma^*$ where $a_1 = \|$, $a_n = \|$, and $a_i \not\in \{\|, \|\}$ for $1 < i < n$, an object of the form $[h, A \rightarrow X_1 X_2 \ldots X_k X_{k+1} \ldots X_m, j]$ is called an item for $w$ if $A \rightarrow X_1 \ldots X_m$ is in $P$, $h$ and $j$ are integers, and $0 \leq j \leq n+1$ [49].
For each integer \( i, 0 \leq i \leq n+1 \), the parser constructs a list of items \( I_i \) such that \([h, A \rightarrow \alpha : \beta, j]\) is in \( I_i \), for \( 0 \leq j \leq i \), if and only if for some \( \gamma \) and \( \delta, Z \Rightarrow^* \gamma A \delta \), \( \gamma \Rightarrow^*_\Gamma \alpha_1 \alpha_2 \ldots \alpha_j \), \( \alpha \Rightarrow^*_\Gamma \alpha_1 \alpha_2 \ldots \alpha_i \), and \( h \) indicates the number of disjoint valid fragments detected in \( \alpha_1 \ldots \alpha_i \) by some sequence of reductions. The sequence of lists \( I_0, I_1, \ldots, I_n \) is called the parse lists of the input string \( w \). It will be the case that \( w \) is in \( L(G') \) if and only if \([h, Z \rightarrow \Sigma, \delta, 0]\) is in \( I_n \). If \( w \) is not in \( L(G') \), there will be an item in \( I_n \) of the form \([h, Z \rightarrow \Sigma \Gamma, \delta] \), where \( h \) is the minimum number of disjoint valid fragments in \( w \). The parsing algorithm basically increments a counter local to each parse being constructed every time a reduction by one of the productions with numbers \( \geq p' \) occurs, which signals the recognition of a locally correct string.

The correctness of Earley's algorithm has been proven (see [3,14]), and as the proof is quite complicated, it will not be duplicated here. The only changes are the addition of the \( h \)-field to each item, which does not significantly change the parsing process as the computation of the \( h \)-field value for each item adds at most a constant number of steps to the process of adding an item to a parse list. Hence, the time bound remains \( O(n^3) \). The modifications increase the space used for an item by a constant amount, so the space bound remains \( O(n^2) \).
Algorithm 3.2 Minimal Local Error Detection

Input. An extended grammar $G' = (N, \Sigma, P, Z)$ constructed by Algorithm 3.1 and a string $w = a_1 a_2 \ldots a_n$ in $\Sigma^*$,
\[a_1 = |-, a_n = -|, a_i \notin \{|-,-|\}, 1 < i < n.\]

Output. The parse lists $I_0, I_1, \ldots, I_n$.

Method. Items on a list are either new or considered. When first placed on a list, an item is new.

(1) Construct $I_0$.

(a) If $(Z, a) \in P$, add $[0, Z \rightarrow a, 0]$ to $I_0$.

(b) If $[0, A \rightarrow B\beta, 0]$ is a new item in $I_0$, add to $I_0$ all items of the form $[0, B \rightarrow \gamma, 0]$ for $(B, \gamma) \in P$ provided it is not already there.

(c) Repeat step (b) until no new items can be added to $I_0$.

(2) Having constructed $I_0, I_1, \ldots, I_{j-1}$, construct $I_j$, $j \leq n$.

(a) For each $[h, B \rightarrow a.\alpha.\beta, i]$ in $I_{j-1}$ such that $a = a_j$,
add $[h, B \rightarrow a.a.\beta, i]$ to $I_j$.

(b) Let $[h', A \rightarrow a, i]$ be a new item in $I_j$, and let $\ell$ be the production number of $(A, a)$ in $P$. Consider this item by searching $I_i$ for items of the form $[h_1, B \rightarrow \gamma.\alpha.\beta, k]$. For each one found, proceed as follows:
(i) If \( \ell < p' \), then add \([h',B \rightarrow \gamma A.\beta,k]\) to \( I_j \) unless there is already an item of the form \([h_2,B \rightarrow \gamma A.\beta,k]\) there, in which case if \( h' \geq h_2 \) then do not add the item, otherwise delete \([h_2,B \rightarrow \gamma A.\beta,k]\) and add the item.

(ii) If \( \ell \geq p' \), then a fragment has been recognized, so the counter is incremented. Add \([h_1+1,B \rightarrow \gamma A.\beta,k]\) to \( I_j \) unless there is already an item of the form \([h_2,B \rightarrow \gamma A.\beta,k]\) there, in which case if \( h_1+1 \geq h_2 \) then do not add the item, otherwise delete \([h_2,B \rightarrow \gamma A.\beta,k]\) and add the item.

(c) If \([h,A \rightarrow \alpha.\beta,B,i]\) is a new item in \( I_j' \), add to \( I_j \) all items of the form \([h,B \rightarrow \gamma,j]\) for \((B,\gamma)\) in \( P \) unless there is already an item of the form \([h_2,B \rightarrow \gamma,j]\) there, in which case if \( h \geq h_2 \) then do not add the item, otherwise delete \([h_2,B \rightarrow \gamma,j]\) and add the item.

(d) Repeat steps (b) and (c) until no new items can be added to \( I_j \).

(3) Repeat step (2) for all \( j \leq n \).

//
Example 3.6

Using the extended grammar $G'$ of Example 3.5, parse lists
are constructed for the string $w = \text{|- a -|}$ by Algorithm 3.2
as follows:

$I_0 = [0, Z \rightarrow \text{|- S -|}, 0]
[0, Z \rightarrow .L R , 0]
[0, L \rightarrow \text{|-} , 0]
[0, L \rightarrow \text{|- S'}, 0]
[0, L \rightarrow \text{|- S} , 0]

$I_1 = [0, Z \rightarrow \text{|- S -|}, 0] [1, F \rightarrow .S , 1]
[0, L \rightarrow \text{|-} , 0] [1, F \rightarrow .'S , 1]
[0, L \rightarrow \text{|- S'}, 0] [1, F \rightarrow .S' , 1]
[0, L \rightarrow \text{|- S} , 0] [1, F \rightarrow .'S' , 1]
[0, S \rightarrow .A S , 1] [1, F \rightarrow .A , 1]
[0, S \rightarrow .b , 1] [1, F \rightarrow .'A , 1]
[1, Z \rightarrow L.R , 0] [1, 'S \rightarrow .A S , 1]
[0, S' \rightarrow .A S' , 1] [1, 'S' \rightarrow .'A S' , 1]
[0, S' \rightarrow .A , 1] [1, 'A \rightarrow .A a , 1]
[0, A \rightarrow .A a , 1] [1, 'A \rightarrow .a , 1]
[0, A \rightarrow .b , 1]
[1, R \rightarrow .F R , 1]
[1, R \rightarrow \text{|-} , 1]
[1, R \rightarrow .'S -| , 1]
[1, R \rightarrow .S -| , 1]
\[ I_2 = [1, 'A + a. , 1] \]
\[ I_3 = [2, R + -| . , 2] \]
\[ [1, F + 'A. , 1] \]
\[ [3, R + F R. , 1] \]
\[ [1, 'S + 'A.S , 1] \]
\[ [3, Z + L R. , 0] \]
\[ [1, 'S' + 'A.S', 1] \]
\[ [2, R + F.R , 1] \]
\[ [1, S + A S , 2] \]
\[ [1, S + .b , 2] \]
\[ [1, S' + A S' , 2] \]
\[ [1, S' + A , 2] \]
\[ [2, R + .F R , 2] \]
\[ [2, R + .-| , 2] \]
\[ [2, R + .'S -| , 2] \]
\[ [2, R + .S -| , 2] \]
\[ [1, A + .A a , 2] \]
\[ [1, A + .b , 2] \]
\[ [2, F + .S , 2] \]
\[ [2, F + .'S , 2] \]
\[ [2, F + .S' , 2] \]
\[ [2, F + .'S' , 2] \]
\[ [2, F + .A , 2] \]
\[ [2, F + .'A , 2] \]
\[ [2, 'S + .'A S , 2] \]
\[ [2, 'S' + .'A S' , 2] \]
\[ [2, 'A + .'A a , 2] \]
\[ [2, 'A + .a , 2] \]
This algorithm contains an optimization which is unnecessary but serves to limit the size of the parse lists somewhat. Whenever a parse attempts to add an item which already exists with a lower h-field value, the new item is not added. This means, essentially, that when two different parses interpret a suffix of the prefix scanned in the same way but with two different fragments counts, the parse with the least fragments is retained.

As noted by Earley [14], Algorithm 3.2 can be augmented to keep pointers from each item in a list to the one or two items which lead to its placement on the list. This would make the reconstruction of a particular right parse from the parse lists fairly straightforward. In fact, the parse lists would then contain the factored set of derivation trees for all of the parses. The following algorithm is a modified version of Earley's algorithm for constructing a right parse from the parse lists generated by Algorithm 3.2. The modifications involve reconstructing a parse with the minimum number of fragments, along with a simultaneous construction of the corresponding interpretation. These modifications do not affect the $O(n^2)$ time bound established by Earley [14] for his version. The basic approach is to back up through the parse lists tracing out the path which represents the parse with the least number of local errors, and is essentially a duplicate of Peterson's version [49].
Algorithm 3.3 Computation of a Valid Interpretation

Input. The grammar G' from Algorithm 3.1, the parse lists from Algorithm 3.2 and the input string $w = a_1a_2 \ldots a_n$, where $a_1 = |-, a_n = -|$, and $a_i \notin \{ |-, -| \}$ for $1 < i < n$.

Output. A right parse for $w$, $\pi$, deriving a minimum number of locally correct strings, and the valid interpretation, $\sigma$, of $w$ defined by $\pi$.

Method. Initially, $\pi = \varepsilon$ and $\sigma = n$; these are global variables. From the set of items consisting of the items of form $[h,Z+\alpha..,0]$ in $I_n$, select one of the items in which $h$ is minimal. This $h$ is the minimum number of disjoint locally correct strings in $w$ ( $h-1$ is the number of local errors in $w$ ) if $h > 0$, and $h = 0$ if and only if $w$ is in $L(G)$. Let $[h,Z+\alpha..,0]$ be the item picked, and execute the routine $R(0,n,Z)$, where $R(i,j,A)$ is defined as follows:

1. Find an item $[h,A+\alpha..,i]$ in $I_j$ with the smallest value of $h$. Let $\ell$ be the production number of $(A,\alpha)$ in $P$. Set $\pi$ to $\ell$ followed by the previous value of $\pi$. If $\ell \geq p'$ then set $\sigma$ to $i$ followed by the previous value of $\sigma$ ( $i$ is the index of the parse list where recognition of $\alpha$ began ). Suppose $\alpha = X_1 \ldots X_m$.

2. Set $k = m$ and $\ell = j$. 
(3) (a) If $X_k \in \Sigma$, reduce $k$ and $\ell$ by 1.

(b) If $X_k \in N$, find an item $[h, X_k \rightarrow \alpha, r]$ in $I_\ell$ for some $r$ such that $[h', A \rightarrow X_1 \ldots X_{k-1}X_k \ldots X_m, i]$ is in $I_r$ and $h$ is minimal. Execute $R(r, \ell, X_k)$.

(c) Reduce $k$ by 1 and set $\ell$ to $r$.

(4) Repeat step (3) until $k = 0$.

//
Example 3.7

Using the parse lists of Example 3.6, a right parse and the corresponding interpretation of the input are computed here by Algorithm 3.3. The recursive calls to \( R(i,j,A) \) are indicated by indentation, with \( \pi \) and \( \sigma \) carried along. The index to the left of each call indicates the order of execution.

[1] \( R(0,3,Z) \) selects \([3, Z \rightarrow L R., 0]\)
\[
\pi = 12, \sigma = 3, a = LR
\]

[2] \( R(1,3,R) \) selects \([3, R \rightarrow F R., 1]\)
\[
\pi = 13\ 12, \sigma = 3, a = FR
\]

[3] \( R(2,3,R) \) selects \([2, R \rightarrow -. , 2]\)
\[
\pi = 17\ 13\ 12, \sigma = 2\ 3, a = -|
\]

[4] \( R(1,2,F) \) selects \([1, F \rightarrow 'A., 1]\)
\[
\pi = 25\ 17\ 13\ 12, \sigma = 1\ 2\ 3, a = 'A
\]

[5] \( R(1,2,'A) \) selects \([1, 'A \rightarrow a., 1]\)
\[
\pi = 10\ 25\ 17\ 13\ 12, \sigma = 1\ 2\ 3, a = a
\]

[6] \( R(0,1,L) \) selects \([0, L \rightarrow |-. , 0]\)
\[
\pi = 14\ 10\ 25\ 17\ 13\ 12, \sigma = 0\ 1\ 2\ 3, a = |-
\]

This interpretation corresponds to \( \text{\texttt{|- a |}} \).
The original version of Algorithm 3.3 correctly finds a right parse of an input \( w \) and operates in time \( O(n^2) \) where \( n \) is the length of \( w \). The modifications do not affect the ability of Algorithm 3.3 to operate in time \( O(n^2) \), since finding a minimal value of \( h \) can be done in a scan of a parse list simultaneously with finding a particular item in step (3)(b). Since the item chosen in step (3)(b) always has a minimal value of \( h \), it is clear that the parse produced by Algorithm 3.3 will generate a minimum number of disjoint locally correct strings.

**Theorem 3.1**

A parse computed by Algorithm 3.3 defines a valid interpretation of the input string.

**Proof**

The interpretation defined contains a minimum number of local errors.

1. If the interpretation contains zero local errors, then the entire string was parsed without use of any error productions which can only occur if the string is a sentence, so the interpretation is valid.

2. Any substring which is not locally correct must be partitioned by the boundary of at least one adjacent pair of fragments, since the grammar only generates sequences of valid fragments.

3. If the interpretation contains \( k \) local errors, then the input string contains \( k+1 \) locally correct substrings according to the interpretation. Any two adjacent
fragments concatenated cannot be a valid fragment, since there would have been a parse generating this single fragment, as the parsing algorithm computes all possible parses. This contradicts the minimality of the chosen parse.

//

Given a parse and valid interpretation computed by Algorithm 3.3, the corresponding extended parse is constructed in a straightforward manner. A detailed algorithm will not be presented. The method is basically to construct a sequence of maximal valid fragments by expanding only nonterminals which are not in the original vocabulary (this includes the prototype of the extended grammar). The sequence of local parses can be constructed by eliminating those production numbers not between 1 and p. The sequences are segmented according to the sequence of fragments defined by the top level structure of the derivation tree.
3.4 Extended Parsing in Perspective

By assuming only deletion errors occur in some string supplied to the parser, a method of error detection was developed which defines the number of errors in a string unambiguously. Although this, in itself, is not an especially good method of error diagnosis, it does allow an approach to parsing with error detection which can significantly reduce the amount of work required to achieve reasonable error diagnosis. The potential points in a string requiring some sort of repair can be determined, and the locally correct segments of the string between these problem points can be reduced to yield a compact representation of the incorrect phrase structure of the string. If the errors are far enough apart, these locally correct segments will likely be parsed correctly, or almost correctly (although this depends on the structural redundancy of the language). In any event, the original input string can be recovered locally as needed for error analysis.

The algorithm used to compute an extended parse of some string in this chapter is rather expensive in time and space compared to the linear time and space bounds of most ad hoc error detection schemes. The algorithm is not quite as expensive as Peterson's minimum distance error detection algorithm, since error productions never have empty right sides so there are fewer possible parses of any given input string. Peterson's algorithm, however, simultaneously
computes a minimum distance correction, while the extended parsing algorithm only detects errors, so the difference is to be expected.

The intent of this chapter has been to formulate the model of extended parsing and demonstrate that an extended parse can be computed (albeit at a high cost). In Chapter 4, the ideas presented in this chapter are used to develop a method of constructing parsers which operate in linear time and compute a reasonable approximation to an extended parse of any string. Thus, the results of this chapter should be viewed as the theoretical basis for the heuristic approach described in the next chapter.
4.1 Introduction

This chapter presents a method of incorporating the error detection technique of extended parsing into practical (by which is meant "linear time and space") bottom-up deterministic parsing algorithms. Section 4.2 contains an analysis of the context condensation error recovery methods of Graham and Rhodes [21] and Druseikis and Ripley [13] in terms of the extended parsing model and algorithm of Chapter 3. This analysis immediately suggests several enhancements of these two techniques. Section 4.3 discusses in detail the formalization of Druseikis and Ripley's simple recovery strategy as extended LR(k) parsing, followed by a discussion of the size of these parsers in section 4.4. Section 4.5 suggests various heuristic techniques for improving the performance of these parsers, and section 4.6 briefly discusses optimizations which can be applied to these parsers.
4.7 Context condensation

Graham and Rhodes [21] demonstrate the utility of backward and forward context condensation moves with a recovery strategy requiring repair of each error as it is detected. Each error is localized as much as possible within a condensed "error phrase" enclosing the detection point. The context represented by this error phrase then allows a pattern matching algorithm to discover a reasonable local repair of the error based on heuristic information supplied by the implementor.

This strategy may be described in terms of the extended parsing model as follows. The modified simple precedence parsing algorithm used guarantees that some suffix of the stack is always a valid fragment by continually verifying that the current potential right side prefix at the top is always a valid right side prefix. This may be done, as described in [21], by sorting the production table lexicographically by the symbols in the right sides and maintaining a pointer to this table during parsing. Since all right sides with a common prefix will be grouped together, it becomes a simple task to verify, when the prefix at the top is to be extended (i.e., when the \( \preceq \) precedence relation holds), that a right side exists with that prefix followed by the symbol to be stacked. This pointer must be saved each time a new right side is begun and set to the first right side whose leftmost symbol is the symbol to be
stacked. Whenever the end of the current potential right side is reached, the pointer points to the production to be reduced by, if there are no errors. When an error is detected, the initial backward condensation move attempts to reduce this prefix by assuming that the end of the potential right side has been located. This reduction process continues until no further reductions can be made, i.e., until the top fragment is a maximal valid fragment. The entire stack is not necessarily a maximal valid fragment, since simple precedence parsers do not detect errors in a strictly left-to-right fashion. Some errors only become apparent after a phrase has been reduced to some nonterminal.

When the backward condensation phase terminates, the forward condensation move assumes that a potential right side starts at the error detection point. The parser proceeds forward until another error is detected, which may be due to detecting either an actual error or a proper infix of some right side. The top fragment condensed by the forward move may be only a single terminal symbol if the input, following the error detection point is a proper infix of some phrase of the language because of the right side verification process, since there may be no rules in the grammar which allow the symbol to start some phrase. In other cases, the top fragment condensed may not be a valid fragment due to the noncanonical detection of errors by
the parsing algorithm. Hence, the right context of an error as detected by this parser is in some cases only an approximation to a valid fragment. Upon termination of the forward condensation phrase, a second backward move is initiated, so that if the top fragment is a valid fragment, it is a maximal valid fragment of the string it represents since all possible reductions are made to both ends. In any case, some prefix of the top fragment is a valid fragment, and hence must be a maximal valid fragment by virtue of the restrictions on the precedence relations. Example 4.1 illustrates the backward and forward condensation actions.

Graham and Rhodes' recovery method is able to detect a local error in an input string and localize it between two maximal valid fragments. However, the method is not necessarily able to exactly determine the extent of the maximal valid fragments on either side of the error detection point, and the error is not necessarily within the leftmost local error interval in the input. The recovery strategy proposed is to immediately use the left and right condensed contexts in a pattern matching process. However, since the condensed contexts do not always represent locally correct strings, the performance of the error analysis may suffer somewhat by not recognizing the existence of another local error in the vicinity of the one undergoing repair. This is partially taken into consideration by only using the suffix of the stack below the error.
Example 4.1

Consider the following simple precedence grammar G (from Aho and Ullman [3], p. 558):

1. $E \rightarrow A$
2. $B \rightarrow E )$
3. $E \rightarrow E + A$
4. $T \rightarrow F$
5. $A \rightarrow T$
6. $T \rightarrow T * F$
7. $F \rightarrow ( B$
8. $F \rightarrow a$

The precedence matrix is not shown here, but may be found in [3], p. 559.

The string $(a + + a)$ contains a syntax error according to this grammar, which would be detected as a symbol pair error when the parser reaches the configuration:

\[
\begin{array}{c|c}
\text{stack} & \text{input} \\
\hline
\$ & ( E + ? \\
E + & + a ) \\
+ & $ \\
\end{array}
\]

At this point, Graham and Rhodes method can make no further reductions on the stack (backward move) since there is no right side at the top; furthermore, the forward move immediately halts with an invalid right side error, since +
Example 4.1 (continued)

is not a prefix of any right side, in the following configuration:

\[
\begin{array}{c|c}
\text{stack} & \text{input} \\
\hline
$( E + ? + ?$ & $a )$ \\
\end{array}
\]

The string $a + a a + )$ contains at least two syntax errors according to $G$. The first is detected as a symbol pair error when the parser reaches the configuration:

\[
\begin{array}{c|c}
\text{stack} & \text{input} \\
\hline
$E + a ?$ & $a + a )$ \\
\end{array}
\]

At this point, the backward move can reduce the stack as follows:

\[
\begin{array}{c|c}
\text{stack} & \text{input} \\
\hline
$E ?$ & $a + a )$ \\
\end{array}
\]

The forward move can then reduce the remaining input to

\[
\begin{array}{c|c}
\text{stack} & \text{input} \\
\hline
$E ? B$ & $\$ \\
\end{array}
\]
point which is a right side prefix for the left context. Of course, the heuristic pattern matching process can be made to cope with such details.

The extended parsing model and algorithm suggests immediate extensions to the modified simple precedence parser discussed above. By use of suffix error productions, the right side verification process will not cause immediate termination of the forward condensation phrase, thereby extending the right context in those cases where the input following the error point is not a prefix of some phrase. Furthermore, if the forward condensation phase is allowed to make reductions by these productions, the right context can be extended farther to the right than is possible with only the original productions. For instance, where a single missing IF would cause the forward move to terminate at the THEN, suffix productions would allow it to continue indefinitely, gathering extended context to give more weight to any repair attempted. Although a straightforward application introduces some problems, since the extended grammar may not be backwards deterministic, various heuristic solutions are possible. The obvious approach is to allow reduction by those suffix error productions which are unique, and otherwise to terminate the forward move.

Extension of this simple precedence technique to preform extended parsing (extended simple precedence parsing) is also possible, but will not be discussed here; instead, the
remainder of this chapter is concerned with the construction of extended LR(k) parsers. The techniques described suggest approaches to solving the problems of nondeterminacy when modifying any deterministic parsing algorithm to perform this generalized error detection process.

The LR(k) parsers present special problems to a straightforward application of the techniques of context condensation used by Graham and Rhodes. The next move of an LR(k) parser can depend on the entire correct prefix already processed (the correct prefix property), making it impossible in many cases to start up the parser after the error detection point. In contrast, the next move of a simple precedence parser depends only on the top of the stack and the next symbol (the locality property).

Despite the difficulties, Druseikis and Ripley [13] and more recently Pennello and DeRemer [48] have reported techniques for grafting a forward context condensation move capability onto the LR(k) parsers. Pennello and DeRemer construct a forward context parser from the finite control of the base parser, while Druseikis and Ripley generate the forward context parser as an integral part of a generalized parsing automaton directly from the grammar by employing a simple extension to DeRemer's method of constructing reduced LR(k) parsers (SLR(k) and LALR(k))[11]. This latter approach is the more appealing since it admits optimization of the combined finite control of the
base and forward context parsers directly during construction, whereas Pennello and DeRemer's method requires that optimizations be performed after construction. Furthermore, a straightforward extension of Druseikis and Ripley's method for the canonical LR(k) construction yields essentially the same results as the method of Pennello and DeRemer. The discussion which follows in this section is phrased in terms of the forward move construction of Druseikis and Ripley.

The forward move is initiated by entering a restart recovery state (or just restart state) determined by the particular input symbol following the error point. The successor recovery states of a restart state are computed by the normal LR(0) construction algorithm; recovery state item sets and item sets in the collection of LR(0) item sets determined by the grammar which are identical are merged. The LR(0) item set corresponding to a restart state will be referred to as a restart item set.

The basis of a restart item set for a symbol X of the grammar (terminal or nonterminal) is the union of the bases of all item sets from the base parser which have an X-successor. Thus, a restart state for X is equivalent to the union of all of the states in the base parser which have an X-successor. Since a restart state for X is only entered when a configuration of the parser has no X-transition, many of the transitions from the restart state are inaccessible. This occurs because the basis set may
contain items with nonterminals to the right of the marker which cause the closure of the basis set to contain items defining transitions for nonterminals which do not start or are not started by $X$, and transitions for terminals which cannot start $X$ (or are not equal to $X$ if $X \in \Sigma$). If the inaccessible transitions are eliminated, the restart states then have mutually disjoint terminal transitions, and nonterminal transitions only for those nonterminals which can be started by or start the particular restart symbol $X$ for each restart state. Furthermore, it becomes immediately apparent that the restart states can be merged into a single restart state without affecting performance. Reduction to some left side $A$ during forward context parsing which uncovers a restart state for any symbol $X$ implies that a right side defining $A$ started in that restart state, so the basis set must have had all possible items with a nonterminal to the right of the marker which could be started by $A$. This is not violated by merging the restart states, and simplifies the the recovery strategy required while reducing somewhat the number of states required in the finite control.

This modified recovery strategy may be described as follows in the terms of the extended parsing model. The restart state assumes no left context at an error detection point, or equivalently, assumes all valid left contexts. The forward condensation phase then attempts to recognize a prefix of the input string following the error point as a
locally correct string which can follow any valid prefix of the grammar. The assumption of any correct prefix is the trick which accomplishes restarting the LR(k) parser after the error. The restart state is essentially a new "initial" state in which recognition of any sentence suffix may begin (nondeterministically). The standard LR(k) parsing algorithm guarantees that any prefix of the remaining input string processed is a valid fragment, and that any reductions which can be made within this prefix will be made. Therefore, when the forward condensation phase terminates, the condensed right context is always a valid fragment and may sometimes be a maximal valid fragment. The forward move terminates when another error is detected, when a reduction of a right side is attempted which would extend across the most recent restart point within the stack, or when a nondeterministic configuration is reached. Reduction across a restart point occurs when a suffix of some right side is recognized; nondeterministic configurations occur where additional context is needed to disambiguate the next move, but is not available due to the restart point on the left and restricted lookahead on the right. Notice that some recovery states may be inaccessible if they are only enterable from nondeterministic configurations.
This recovery method is able to detect a local error within the leftmost local error interval in the input string because of the correct prefix property (in fact, the rightmost boundary of the leftmost local error interval is detected). This local error is localized between two valid fragments, and in some cases (if default reductions are allowed at the error point) these may be maximal valid fragments. The heuristic recovery strategy employed by Druseikis and Ripley is to restart each time the forward condensation phase terminates, where in the case that a unique reduction across the restart point occurs, the left side reduced to is used to restart rather than the next input symbol. Thus, this strategy essentially computes an interpretation (determined by the restart points) of the input string which may sometimes, but not in general, be a valid interpretation. The interpretation is not guaranteed to be valid because nondeterministic decisions and reductions across restart points force additional restarts. Example 4.2 illustrates an SLR(1) parser with recovery states for the augmented grammar of Example 3.5, and example 4.3 illustrates the recovery process.

This "recover by restarting" technique can be applied to the canonical LR(k) parser construction in a straightforward manner. Since the restart state assumes all valid left contexts, for any LR(0) item \([A \rightarrow \alpha.X\beta]\) in the restart item set, any element of \(\text{FOLLOW}_k(A)\) is equally possible, which is the default for all items in the SLR(k) construction. Therefore,
Example 4.2

Let the augmented grammar G be defined by the productions

1. \( Z \rightarrow \text{--} S \text{--} \)
2. \( S \rightarrow A S \)
3. \( S \rightarrow b \)
4. \( A \rightarrow A a \)
5. \( A \rightarrow b \)

The item sets \( I_0 \) through \( I_7 \) shown below determine an SLR(1) parser; the restart item set is \( I_8 \), and \( I_9 \) is the only recovery item set which cannot be merged with the base parser. Note that \( I_9 \) is nondeterministic for lookahead \( \text{--} \).

\[ I_0 \quad Z \rightarrow \text{--} S \text{--} \quad \text{shift} \quad \text{--} \quad \text{goto} \ 1 \]

\[ I_1 \quad Z \rightarrow \text{--} S \text{--} \quad \text{goto} \ 2 \]
\[ S \rightarrow A S \quad \text{goto} \ 3 \]
\[ S \rightarrow b \quad \text{shift} \ b \quad \text{goto} \ 4 \]
\[ A \rightarrow A a \quad \text{goto} \ 3 \]
\[ A \rightarrow b \quad \text{shift} \ b \quad \text{goto} \ 4 \]

\[ I_2 \quad Z \rightarrow \text{--} S \text{--} \quad \text{shift} \quad \text{--} \quad \text{goto} \ 5 \]
Example 4.2 (continued)

\[ I_3 \; S \to A.S \]
\[ \quad A \to A.a \]
\[ \quad S \to .A S \]
\[ \quad S \to .b \]
\[ \quad A \to .A a \]
\[ \quad A \to .b \]

\[ \text{goto 6} \]
\[ \text{shift A goto 7} \]
\[ \text{goto 3} \]
\[ \text{shift b goto 4} \]
\[ \text{goto 3} \]
\[ \text{shift b goto 4} \]

\[ I_4 \; S \to b. \]
\[ \quad A \to b. \]

\[ \text{reduce if \{ -| \}} \]
\[ \text{reduce if \{ a, b \}} \]

\[ I_5 \; Z \to |-S-.| \]

\[ \text{accept} \]

\[ I_6 \; S \to A S. \]

\[ \text{reduce if \{ -| \}} \]

\[ I_7 \; A \to A a. \]

\[ \text{reduce if \{ a, b \}} \]

\[ I_8 \; Z \to .|-S-| \]
\[ \quad Z \to |.-S-.| \]
\[ \quad Z \to |.-S-.| \]
\[ \quad S \to .A S \]
\[ \quad S \to A.S \]
\[ \quad S \to .b \]
\[ \quad A \to .A a \]
\[ \quad A \to A.a \]
\[ \quad A \to .b \]

\[ \text{shift |- goto 1} \]
\[ \text{goto 9} \]
\[ \text{shift -| goto 5} \]
\[ \text{goto 3} \]
\[ \text{goto 9} \]
\[ \text{shift b goto 4} \]
\[ \text{goto 3} \]
\[ \text{shift a goto 7} \]
\[ \text{shift b goto 4} \]
Example 4.2 (continued)

I9  Z + |− S.−|  
     S + A S.  
     shift −| goto 5  
     reduce if { −| }

Parsing Table

<table>
<thead>
<tr>
<th></th>
<th>−</th>
<th>a</th>
<th>b</th>
<th>−</th>
<th>ε</th>
<th>S</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>S4</td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>S5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S7</td>
<td>S4</td>
<td></td>
<td></td>
<td>6</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>R(5)</td>
<td>R(5)</td>
<td>R(3)</td>
<td></td>
<td>6</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>R(2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>R(4)</td>
<td>R(4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>S1</td>
<td>S7</td>
<td>S4</td>
<td>S5</td>
<td>9</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

goto actions are indicated by integer state numbers
parsing actions are abbreviated as follows:
S   shift   N inadequate transition
R(i) reduce i
A   accept
empty entries denote error
Example 4.3

The recovery strategy is illustrated for the parser shown in Example 4.1 on the string $w = |a b|$. The symbol ? is used to mark each restart point in the stack, and the parser states are represented by integers, and the grammar symbols are shown on the stack for clarity.

<table>
<thead>
<tr>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. 0</td>
<td>- a b -</td>
<td>shift</td>
</tr>
<tr>
<td>2. 0</td>
<td>- 1</td>
<td>a b -</td>
</tr>
<tr>
<td>3. 0</td>
<td>- 1 ? 8</td>
<td>a b -</td>
</tr>
<tr>
<td>4. 0</td>
<td>- 1 ? 8 a 7</td>
<td>b -</td>
</tr>
<tr>
<td>5. 0</td>
<td>- 1 ? 8 A 3</td>
<td>b -</td>
</tr>
<tr>
<td>6. 0</td>
<td>- 1 ? 8 A 3 b 4</td>
<td>-</td>
</tr>
<tr>
<td>7. 0</td>
<td>- 1 ? 8 A 3 S 9</td>
<td>-</td>
</tr>
<tr>
<td>8. 0</td>
<td>- 1 ? 8 A 3 S 9 ? 8</td>
<td>-</td>
</tr>
<tr>
<td>9. 0</td>
<td>- 1 ? 8 A 3 S 9 ? 8 -</td>
<td>5</td>
</tr>
</tbody>
</table>

Notes

1 - lookahead error
2 - reduce $A \rightarrow Aa$ pops the restart state; since the action is unique, recover by restarting with $A$
3 - state 9 is inadequate on lookahead -|

The interpretation of $w = |a b|$ represented by the recovery is $\sigma = 0, 1, 3, 4$ which is not a valid interpretation.
the restart item set for any (variant) LR(k) parser, must be the same, given the same grammar. The only difference is in the successor computation, with the canonical LR(k) method generating more recovery states than the SLR(k) method, in general, which cannot be merged with states in the base parser due to the more discriminating method of lookahead computation.

The extended parsing model and algorithm suggests improvements to this recovery technique, which are explored in detail in the remainder of this chapter. The discussion is phrased in the terms of the canonical LR(k) parser construction, but the results are immediately applicable to reduced versions of LR(k) parsers through the use of the characteristic LR(k) parser construction of Geller and Harrison [19]. The details of this construction are not relevant to this dissertation; the essential point is that reduced versions of LR(k) parsers, such as SLR(k) and LALR(k), can be directly constructed by a generalized version of the canonical LR(k) parser construction algorithm, rather than by first constructing the collection of LR(0) item sets and then adding lookahead to resolve inadequate states.
4.3 Extended LR(k) parsing

The forward context LR(k) parsing techniques mentioned in the previous section suffer in performance because the position of the most recent restart move with respect to the current parser configuration is retained only within the stack. This means that each time the forward context parser makes a reduction, it must examine each state popped from the stack to see if it is the restart state. The reason that the forward context parser cannot remember this simple fact is that the restart item set is a collection of LR(k) items which are not all initial items (where an initial LR(k) item is an LR(k) item of the form \([A\cdot+a,u]\)). That is, the restart state begins the recognition of a right side at all positions in the middle (as well as at the front), as if the parser had already recognized a correct prefix ending with the corresponding right side prefix.

The efficiency of the forward context parsing algorithm can be improved by the use of suffix error productions in the construction of the restart item set. The cost of this improvement is an increase in the number of unique recovery states, due to the new productions added. Using only initial items in the restart item set, reduction by a suffix error production signals that some reduction would have occurred across the most recent restart point. Thus, the parsing algorithm no longer need examine the stack at each reduction in order to determine when to halt the forward condensation phase,
since it knows when a reduction across the most recent restart point occurs, although not necessarily which one. Furthermore, it may be possible in some cases to make a deterministic reduction by a suffix error production where before the forward move would have terminated. Thus, the forward context may sometimes be longer, since the lookahead symbol causing the reduction may eventually be shifted onto the stack; (see examples 4.4 and 4.5).

A forward condensation move made by a parser constructed as described with suffix error productions terminates when another local error is detected or when a nondeterministic configuration is reached, which occurs whenever two or more production suffixes give different interpretations to the same valid fragment. The use of suffix error productions enables the forward context parser to achieve a more formal forward move than previous methods. The simple recovery strategy suggested by Druseikis and Ripley in [13] employs a few simple heuristics which allow their parser to continue after halting. In the case that a halt occurs due to a reduction across a restart point, the parser is restarted with the left side reduced if the reduction was unique. This heuristic becomes a formal parsing action when reductions by suffix productions are employed. Furthermore, in cases where the reduction is not unique in Druseikis and Ripley's parser due to a left side having two right sides with the same suffix, the use of suffix productions results in a unique suffix reduction. Trapped states (by which the authors
Example 4.4

The extra suffix productions, derived as in Algorithm 3.1, are shown below for the grammar of Example 4.2.

6. 'Z → 'S - |
7. 'Z → - |
8. 'Z → S - |
9. 'S → 'A S
10. 'A → 'A a
11. 'A → a

Item sets I₀ through I₇ are the same as in Example 4.2; the recovery item sets which cannot be merged are shown here. I₈ is the restart item set as before. Note that there are no inadequate states in this particular parser; in general this will not be the case.

I₈  Z → . | - S - |
    'Z → .|'S - |
    'Z → .- |
    'Z → . S - |
    S → .A S
    'S → .|'A S
    S → .b
    A → .A a
    'A → .|'A a
    'A → .a
    A → .b

    shift | - goto 1
    goto 9
    shift - | goto 10
    goto 11
    goto 3
    goto 12
    shift b goto 4
    goto 3
    goto 12
    shift a goto 13
    shift b goto 4
Example 4.4 (continued)

I_9  'Z → 'S.-|  

I_10  'Z → -|.  

I_11  'Z → S.-|  

I_12  'S → 'A.S  

'A → 'A.a  

S → .A S  

S → .b  

A → .A a  

A → .b  

I_13  'A → a.  

I_14  'Z → 'S -|  

I_15  'Z → S -|  

I_16  'S → 'A S.  

I_17  'A → 'A a.  

shift -| goto 14

error accept

shift -| goto 15

goto 16

shift a goto 17

goto 3

shift b goto 4

goto 3

shift b goto 4

reduce if { a, b }

error accept

error accept

reduce if { -| }

reduce if { a, b }
Example 4.4 (continued)

Parsing Table

<table>
<thead>
<tr>
<th></th>
<th>&lt;-</th>
<th>a</th>
<th>b</th>
<th>&lt;-</th>
<th>e</th>
<th>S</th>
<th>'S</th>
<th>A</th>
<th>'A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S1</td>
<td></td>
<td></td>
<td></td>
<td>ε</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>S4</td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>S5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>S7</td>
<td>S4</td>
<td></td>
<td>6</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>R(5)</td>
<td>R(5)</td>
<td>R(3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>R(2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>R(4)</td>
<td>R(4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>S1</td>
<td>S13</td>
<td>S4</td>
<td>S10</td>
<td>11</td>
<td>9</td>
<td>3</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>S14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>EA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>S15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>S17</td>
<td>S4</td>
<td></td>
<td></td>
<td>16</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>R(11)</td>
<td>R(11)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td>EA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td>EA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td>R(9)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>R(10)</td>
<td>R(10)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

EA denotes error accept
Example 4.5

The recovery of the parser in Example 4.4 is illustrated with the input string $w \equiv | - a b - |$ of Example 4.3.

<table>
<thead>
<tr>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. 0</td>
<td>$</td>
<td>- a b -</td>
</tr>
<tr>
<td>2. 0</td>
<td>$</td>
<td>- 1 a b -</td>
</tr>
<tr>
<td>3. 0</td>
<td>$</td>
<td>- 1 ? 8 a b -</td>
</tr>
<tr>
<td>4. 0</td>
<td>$</td>
<td>- 1 ? 8 a 13 b -</td>
</tr>
<tr>
<td>5. 0</td>
<td>$</td>
<td>- 1 ? 8 'A 12 b -</td>
</tr>
<tr>
<td>6. 0</td>
<td>$</td>
<td>- 1 ? 8 'A 12 b 4 -</td>
</tr>
<tr>
<td>7. 0</td>
<td>$</td>
<td>- 1 ? 8 'A 12 S 16 -</td>
</tr>
<tr>
<td>8. 0</td>
<td>$</td>
<td>- 1 ? 8 'S 9 -</td>
</tr>
<tr>
<td>9. 0</td>
<td>$</td>
<td>- 1 ? 8 'S 9 -</td>
</tr>
</tbody>
</table>

Notes

$^1$ - lookahead error

The interpretation of $w \equiv | - a b - |$ represented by this recovery is $\sigma = 0, 1, 4$ which is a valid interpretation. The parser of Example 4.3 did not determine a valid interpretation.
mean inadequate LR(0) recovery states which cannot be resolved by DeRemer's simple lookahead technique [11]) do not occur using the LR(k) construction technique, and hence the need for "dynamic resolution" disappears. This refers to dynamically examining the item set during parsing to see if the action called for by a particular lookahead symbol is unique. The only ambiguous transitions are those with conflicting actions for a given lookahead, which cannot be resolved in this dynamic manner. If there are not any states in the parser which are inadequate, the forward context parser will always either reach the next local error in the string or reach the end of the string after an error is detected. In general, however, there will be inadequate states in the finite control.

The construction described yields a parser which, if used with the simple recovery strategy of Druseikis and Ripley, determines an interpretation of any input string (with endmarkers) which can be a reasonable approximation of a valid interpretation. If the LR(k) parsing algorithm is modified slightly to incorporate the restart mechanism and to suitably mark the parse list and stack, the resulting parser is an approximation to an extended parser. The general ideas presented above are now stated formally, as extensions to the LR(k) parsing algorithm presented by Aho and Ullman [3]. The term "state" is commonly used in place of "table" as defined in Definition 4.1.
Definition 4.1

An extended LR(k) parser for an augmented CFG $G=(N, \Sigma, P, Z)$ is a three-tuple $(T', T_0, T_1)$, where $T$ is a set of extended LR(k) tables, $T_0$ is the distinguished initial table, and $T_1$ is the distinguished restart table. Each table $T \in T$ is a pair of functions $(f_T, g_T)$ such that

1. $f_T$, the parsing action function, is a map from $\Sigma^k$ into $\{\text{shift, restart, error, accept, error accept}\} \cup \{\text{reduce $\rho$} \mid \rho \in P \cup P_S\}$ where $P_S$ is a set of suffix error productions of $G$.

2. $g_T$, the goto function, is a map from $\mathcal{V} \cup \mathcal{N}_S$ into $\mathcal{T}_u \cup \{\text{error}\}$, where $\mathcal{N}_S$ is the set of suffix nonterminals defined in $P_S$.

An extended LR(k) parser behaves as a shift-reduce parsing algorithm. At the start, the stack contains a bottom marker and the initial table $T_0$ and nothing else. Whenever a parsing action is restart or error, and whenever a goto is error, the restart table, $T_1$, is stacked and parsing continues. The error transitions into $T_1$ are indications of true local errors, whereas the restart transitions are indications of nondeterministic configurations. The construction of the tables for a canonical extended LR(k) parser is basically the same as the construction of the tables for a canonical LR(k) parser, with the addition of a restart table. The construction is presented in Algorithm 4.1, 4.2, and 4.3, which assume that a set of LR(k) items is computed by Algorithm
Algorithm 4.1 Extending a Grammar for Recovery

Input. An augmented CFG \( G = ( N', \Sigma', P', Z ) \).

Output. The extended grammar \( G' = ( N, \Sigma, P, Z ) \)
which includes all suffix error productions of \( G \),
and the integer \( p \). Productions numbered 1 to \( p \)
will be from \( P' \).

Method. (The method is basically Algorithm 3.1 for suffix
productions only.) Let \( P_s \), the set of suffix error
productions, be initially empty. Let the productions
in \( P' \) be numbered 1 to \( p \).

1. For each \((A, \alpha)\) in \( P' \) with \( \log(\alpha) = n > 0 \), set
   \[ P_s = P_s \cup \{ ('A, \alpha[1:n]), 1 < i \leq n \}. \]
2. Eliminate from \( P_s \) all single productions of the form
   \((A, A)\) and all productions of the form \((A, \alpha)\) such that
   \((A, \alpha) \in P'\).
3. For each \((A, \alpha)\) in \( P_s \) such that \( \alpha = B\beta \) for some
   \( \beta \in (V')^* \), \( B \in N' \), add \((A, 'B\beta)\) to \( P_s \).
4. For each \((A, \alpha)\) in \( P' \) such that \( \alpha = B\beta \) for some \( \beta \in (V')^* \),
   \( B \in N' \), add \((A, 'B\beta)\) to \( P_s \).
5. Let \( G' = (N, \Sigma, P, Z) \) be the CFG defined by
   \[ N = N' \cup \{ 'A \mid A \in N' \} \]
   \[ \Sigma = \Sigma' \]
   \[ P = P' \cup P_s \]
such that the productions of \( P' \) remain
   numbered 1 to \( p \).

//

Note that this grammar is not reduced.
**Algorithm 4.2** The set of extended LR(k) restart items

**Input.** An extended CFG $G' = (N, \Sigma, P, Z)$ computed by Algorithm 4.1, and an integer $k$.

**Output.** $R_k(\epsilon)$, the set of LR(k) restart items for $G'$.

**Method.** $R_k(\epsilon)$ is initially empty.

For each $(A, \alpha)$ in $P$, add $[A \cdot \alpha, u]$ to $R_k(\epsilon)$ for all $u \in \text{FOLLOW}_k(\alpha)$.

//
Algorithm 4.3 Collection of sets of extended LR(k) items

Input. An augmented LR(k) grammar $G = (N', \Sigma', P', Z)$ and an integer $k$.

Output. $\Omega_k'$, the collection of sets of extended LR(k) items for $G$, and $G' = (N, \Sigma, P, Z)$, the extended grammar.

Method.

(1) Construct the collection of sets of valid LR(k) items for $G$ by Algorithm 5.9 of Aho and Ullman [3], and denote the collection by $\Omega_k$.

(2) Construct the extended grammar $G'$ from $G$ by Algorithm 4.1.

(3) Place $R_k(\epsilon)$, the set of extended LR(k) restart items, in $\Omega_k$. The set $R_k(\epsilon)$ is initially unmarked.

(4) If a set of items $V_k(\gamma)$ in $\Omega_k$ is unmarked,
   (a) compute, for each $X \in V$, $V_k(\gamma X)$.
   (b) If $V_k(\gamma X) \neq \emptyset$, then add $V_k(\gamma X)$ to $\Omega_k$ as an unmarked set of items if it is not already there.
   (c) Mark $V_k(\gamma)$.

(5) Repeat step (4) until all sets of items in $\Omega_k$ are marked.

//
5.8 of Aho and Ullman [3], replacing their definition of "augmented grammar" by Definition 3.7 of Chapter 3. The tables of an extended LR(k) parser are then determined as given by Definition 4.2.

**Definition 4.2**

Let \( G = (N, \Sigma, P, \Lambda) \) be a CFG and let \( \Omega_k \) be a collection of sets of extended LR(k) items for \( G \) as computed by algorithm 4.3. For each \( I \in \Omega_k \), define \( T = T(I) \), the table associated with \( I \), as the pair of functions \((f_T, g_T)\) defined by

1. \( f_T(u) = \text{shift} \) if \([A \rightarrow \beta_1 \beta_2 \Lambda] \) is in \( I \), \( \beta_2 \neq \varepsilon \), and \( u \in \operatorname{EEFF}_k(\beta_2 \Lambda) \).
2. \( f_T(u) = \text{reduce} \ \rho \) if \([A \rightarrow \beta \Lambda, u] \) is in \( I \) and \( \rho = (A, \beta) \in P \).
3. \( f_T(\varepsilon) = \text{accept} \) if \([Z \rightarrow \alpha \varepsilon] \) is in \( I \).
4. \( f_T(\varepsilon) = \text{error accept} \) if \(['Z \rightarrow \alpha, \varepsilon] \) is in \( I \).
5. if \( f_T(u) \) has more than one definition under rules (1) - (4), then \( f_T(u) = \text{restart} \); if \( f_T(u) \) is left undefined by rules (1) - (4), then \( f_T(u) = \text{error} \).
6. \( g_T(x) = T(\text{goto}(I, x)) \) if \( \text{goto}(I, x) \neq \phi \).
7. \( g_T(x) = \text{error} \) if \( \text{goto}(I, x) = \phi \).

The parser of Example 4.4 was constructed by this method.
A more extensive example is found in Appendix I and II for the grammar shown below. Appendix I contains the SLR(1) item sets for this grammar, and Appendix II contains the recovery item sets which are disjoint from the base parser's item sets.

An SLR(1) Grammar

block → begin corral end

\[\text{corral → decl-list \text{seicolon} stmt-list | stmt-list}\]

\[\text{decl-list → decl | decl-list \text{seicolon} decl}\]

\[\text{stmt-list → stmt | stmt-list \text{seicolon} stmt}\]

\[\text{decl → type def-list}\]

\[\text{def-list → def | def-list comma def}\]

\[\text{def → id | id = sexpr}\]

\[\text{stmt → block | id + expr}\]

\[\text{expr → sexpr | sexpr = sexpr}\]

\[\text{sexpr → id | sexpr + id}\]

sentence → | - block - |
A sample parse of a string using the parser of Appendix I and II is shown in Example 4.6, according to the extended parsing algorithm given in Algorithm 4.4. The special symbol ? marks a local error point in the stack and in the parse list, and the special symbol # marks a nondeterministic restart point.

Since the finite control of an extended LR(k) parser contains the finite control of a canonical LR(k) parser for the language as a subset, correct strings are processed in linear time and space producing a right parse; since the restart state is never entered, there are no reductions by suffix productions for a correct input. Furthermore, incorrect strings are also processed in linear time and space; the space bound is linear since at worst a restart will be required on each input symbol, leaving 2n symbols and 2n+1 tables on the stack; the time bound is linear since the restart state has an X-successor for all XεV, and so the time bound of the canonical LR(k) parsing algorithm is not affected by the addition of the restart facility.

The size of an extended LR(k) parser (in terms of the number of states) is, of course, larger than that of the corresponding LR(k) parser for the same grammar. Before continuing with an analysis of the size of these parsers, two simple optimizations will be mentioned which reduce the number of recovery states somewhat.
Algorithm 4.4 Extended LR(k) parsing

Input. A set of extended LR(k) tables for an augmented LR(k) grammar $G = (N, \Sigma, P, Z)$, with $T_0 \in \Gamma$ designated as the initial table, $T_1 \in \Gamma$ designated as the restart table, and an input string with unique endmarkers $x = |-w-|$.

Output. If $x \in L(G)$, the right parse of $x$ according to $G$, otherwise a sequence of local parses corresponding to the sequence of valid fragments left on the stack.

Method. The stack initially contains $\$$ $T_0$, where $\$$ marks the bottom of the stack. The parse list is initially empty. Perform steps (1) and (2) until acceptance occurs. If the acceptance action is accept, then the parse list contains the right parse of $x$.

1. Let $u$, the lookahead string, consist of the next $k$ input symbols.

2. Apply the parsing action function $f_T$ of the table $T$ on top of the stack to $u$.
   
   (a) If $f_T(u) = \text{shift}$, then the next input symbol, $a$, is removed from the input and is pushed onto the stack followed by $g_T(\varepsilon)$. Return to step (1).
(b) If $f_T(u) = \text{reduce } \rho$ and $\rho = (A,\alpha)$, then $2*\log(\alpha)$ symbols are removed from the top of the stack, and the production number of $\rho$ is placed at the end of the parse list. A new table $T$ is exposed as the top of the stack and $A$ is pushed onto the stack followed by $g_T(A)$. Return to step (1).

(c) If $f_T(u) = \text{error}$, place ? at the end of the parse list and push ? onto the stack followed by $T_1$. Return to step (1).

(d) If $f_T(u) = \text{restart}$, place # at the end of the parse list and push # onto the stack followed by $T_1$. Return to step (1).

(e) If $f_T(u) = \text{accept}$ or $\text{error accept}$, then halt.
Example 4.6

This example illustrates an extended parse of the string

\[- \text{begin} \text{id} = \text{id} + \text{id} \; ; \; \text{id} + \text{id} \text{ end} \; -\]

with the parser specified by the item sets shown in Appendix I and II. The grammar is shown on page 124. This string contains one local error interval consisting of the leftmost \text{id} symbol. In order to avoid clutter, the stack below each restart point is truncated.

<table>
<thead>
<tr>
<th>stack</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $ 0</td>
<td>shift</td>
</tr>
<tr>
<td>2. $ 0</td>
<td>- 1</td>
</tr>
<tr>
<td>3. $ 0</td>
<td>- 1 \text{begin} 2</td>
</tr>
<tr>
<td>4. $ 0</td>
<td>- 1 \text{begin} 2 \text{id} 4</td>
</tr>
<tr>
<td>5. ? 79</td>
<td>shift</td>
</tr>
<tr>
<td>6. ? 79 = 36</td>
<td>shift</td>
</tr>
<tr>
<td>7. ? 79 = 36 \text{id} 31</td>
<td>reduce sexpr + \text{id}</td>
</tr>
<tr>
<td>8. ? 79 = 36 sexpr 56</td>
<td>shift</td>
</tr>
<tr>
<td>9. ? 79 = 36 sexpr 56 + 18</td>
<td>shift</td>
</tr>
<tr>
<td>10. ? 79 = 36 sexpr 56 + 18 \text{id} 32</td>
<td>reduce sexpr + sexpr + \text{id}</td>
</tr>
<tr>
<td>11. ? 79 = 36 sexpr 56</td>
<td>restart\textsuperscript{1}</td>
</tr>
<tr>
<td>12. # 79</td>
<td>shift</td>
</tr>
<tr>
<td>13. # 79 ; 39</td>
<td>shift</td>
</tr>
<tr>
<td>14. # 79 ; 39 \text{id} 4</td>
<td>shift</td>
</tr>
<tr>
<td>15. # 79 ; 39 \text{id} 4 + 9</td>
<td>shift</td>
</tr>
</tbody>
</table>
Example 4.6 (continued)

<table>
<thead>
<tr>
<th>stack</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>16. # 79 ; 39 id 4 + 9 id 31</td>
<td>reduce sexpr (\rightarrow) id</td>
</tr>
<tr>
<td>17. # 79 ; 39 id 4 + 9 sexpr 14</td>
<td>reduce expr (\rightarrow) sexpr</td>
</tr>
<tr>
<td>18. # 79 ; 39 id 4 + 9 expr 30</td>
<td>reduce stmt (\rightarrow) id + expr</td>
</tr>
<tr>
<td>19. # 79 ; 39 stmt 57</td>
<td>restart(^2)</td>
</tr>
<tr>
<td>20. # 79</td>
<td>shift</td>
</tr>
<tr>
<td>21. # 79 end 67</td>
<td>reduce 'block (\rightarrow) end</td>
</tr>
<tr>
<td>22. # 79 'block 41</td>
<td>shift</td>
</tr>
<tr>
<td>23. # 79 'block 41 -</td>
<td>78</td>
</tr>
</tbody>
</table>

Notes

1 - reduce/reduce conflict between

\['\text{def} + = \text{sexpr} \quad \text{and} \quad '\text{expr} + = \text{sexpr}]

2 - reduce/reduce conflict between

\['\text{stmt-list} \rightarrow \text{semicolon} \text{ stmt} \quad \text{and} \quad \text{stmt-list} \rightarrow \text{stmt}]

This conflict occurs because of the suffix production

\['\text{corral} \rightarrow \text{semicolon} \text{ stmt-list}]

The input string is interpreted as the sequence of valid fragments shown below, where the last two restart points are due to nondeterministic configurations. The interpretation is not valid, but the parser interprets the first restart point as a local error and merely notes that it entered an ambiguous configuration for the others.

\[
\text{|- begin id} \uparrow = \text{id} + \text{id} \uparrow ; \text{id} \uparrow \text{id} \quad \text{end} \quad |\]
It is immediately obvious that the initial state and the restart state can be merged into a single state. Since the left endmarker symbol only occurs at the left end of any string, the only time a transition under that symbol can occur is when the parser is initiated, so that transition in the restart state is never accessed. Furthermore, from the initial state, the left endmarker is always the first input symbol, so the other error entries are never accessed. Hence, the initial state and the restart state have disjoint accessible transitions. The construction of an extended LR(k) parser could be rephrased as simply a new definition of the initial LR(k) item set, but this would make the process of testing the base parser for consistency more complicated. With this change, the difference between an initial state, an error restart state and an ambiguous restart state is only apparent from the stack contents immediately below the state. The base parser consists of all states accessible from the initial state following a |- -transition.

The construction of suffix error productions causes the finite control of an extended LR(k) parser to lose information about which particular prefix is missing in some cases. This occurs when a particular left side has two (or more) right sides with the same suffix; only one suffix production is generated. This information could be maintained in the finite control, but it is only useful
during the reparation process and is not necessary during extended LR(k) parsing. Along these same lines, the construction of suffix productions in Algorithm 4.1 still results in more information being maintained in the finite control than is necessary for extended LR(k) parsing. Since error productions are only used in the restart item set, the need to distinguish between the phrases generated by A and the phrases generated by \textasciitilde A is eliminated. This was required by the general algorithm of Chapter 3 because a fragment could start at any point in the input string and suffix productions could only be allowed at the left end of a fragment. During extended LR(k) parsing, fragments can only start after an error (or a nondeterministic configuration). By eliminating the distinction between primed and non-primed nonterminals (and only using suffix productions in the restart state), the number of recovery states which remember the location of a restart point is cut in half.
4.4 The size of an extended LR(k) parser

The size of an LR(k) parser, in terms of the number of states and transitions, depends on the particular grammar from which it was constructed. There are, however, various estimators of the size of a parser, and one is given by the size of the grammar defined as the number of productions plus the sum of the lengths of the right sides of all productions. The form and number of suffix error productions depends entirely on the original grammar and the construction of recovery states is identical with the construction of the states of the base parser. Therefore, the size of an extended LR(k) parser should depend on the particular grammar in a manner similar to the way the size of the base parser depends on the grammar. An estimate of the size of the extended grammar in terms of the original grammar is likely a crude indication of the increase in size of the extended LR(k) parser over the base parser. However, a more precise measure depends on the particular grammar.

If the length of the right side of production \( p_i \) is denoted by \( \ell_i \), and the number of productions is \( p \), then the size of a grammar \( G \) can be written as

\[
S = \sum_{i=1}^{p} (\ell_i + 1).
\]
In the construction of the (optimized) extended grammar as discussed at the end of Section 4.2, each production \( p_i \) of the original grammar contributes \( \ell_i-1 \) suffix productions (there are no \( \varepsilon \)-rules by assumption) of lengths \( 1, 2, \ldots, \ell_i-1 \), so that the total length contributed by production \( p_i \) of the original grammar to the extended grammar can be written as

\[
L_i = \ell_i + \sum_{j=1}^{\ell_i} j = \ell_i + \frac{\ell_i}{2}(\ell_i + 1) = \frac{1}{2}(\ell_i^2 + 3\ell_i).
\]

Thus, the size of the extended grammar \( G' \) can be written as

\[
s' = \sum_{i=1}^{p} L_i = \frac{1}{2} \sum_{i=1}^{p} (\ell_i^2 + 3\ell_i).
\]

If the bound on the length of a right side of \( G \) is given by \( k > 0 \), then

\[
s \leq p(k+1) \quad s' \leq \frac{1}{2}p(k^2+3k)
\]

This gives

\[
p \sim \frac{s}{k+1}
\]

so that

\[
s' \sim \frac{1}{2} \frac{s}{k+1}(k^2+3k) = \frac{1}{2}s(k+\frac{2k}{k+1}) \quad \text{or} \quad s' \sim \frac{1}{2}ks.
\]

The size of the extended grammar is asymptotically less than \( \frac{1}{2}k \) times the size of the original grammar. This bound is never reached, although a grammar in Chomsky Normal Form comes close to it. This indicates that an extended LR(k) parser is probably larger than its base parser by at most a constant factor depending on the original grammar. For example, the extended SLR(1) in Appendix II (if primed and non-primed nonterminals are identified as suggested in section 4.3) is only about twice the size of the base SLR(1) parser.
4.5 Resolution of Nondeterminism

Although the extended LR(k) parser is able to detect many local errors, it may not be able to detect all of them due to the interference caused by inadequate states. Some local errors may be "mislabelled" as restart points caused by nondeterministic configurations of the parser. Such ambiguous situations are caused by the rather simplistic method used by the restart recovery technique to simulate all possible interpretations of the input following a local error. Local parsing can continue uninterrupted for only as long as all interpretations which have a next move have the same next move with respect to the original grammar. Thus, on reaching the end of a non-sentence, the sequence of valid fragments detected is only an approximation to the extended parse of the string as defined in Chapter 3. This approximation compares unfavorably to a true extended parse both because it is not guaranteed of detecting all of the local errors and also because it cannot determine in all cases a maximal valid fragment of the locally correct string between two local errors. This latter deficiency results in the parse stack possibly becoming quite large for reasonably sized incorrect programs. In order to reduce the possibilities for ignoring local errors and to increase the number of reductions that can be made on the stack, various heuristic techniques are suggested in this section for eliminating or delaying the occurrence of a nondeterministic configuration.
The obvious approach is to try to resolve the conflict by additional lookahead, but as noted by Druseikis and Ripley [13], this will not help in many cases. In Example 4.6, no amount of lookahead can resolve the conflict between reducing to 'def or to 'expr, in step II of the parse. However, since it is impossible in such cases to make a decision without left context, which is not available until the reparation phase is entered, any choice from among structurally equivalent possibilities will suffice to enable the extended parser to continue with some arbitrary interpretation of the structure of the valid fragment. One way to resolve inadequate recovery states, then, is to simply make a heuristic decision during parser construction as to which reduction to take from among a set of conflicting but equivalent reductions. The heuristic should select the left side which will lead to a configuration which allows the largest set of following phrases. For instance, in Example 4.6 again, choosing the reduction to 'def in item set I_{56} of Appendix II (shown below) is preferrable, since then any phrase which can follow a semicolon will be accepted. Choosing the reduction to 'expr would cause a local error to be detected if a declaration followed the semicolon, which violates the notion of a local error.

\[
\begin{align*}
I_{56} & \text{ sexpr} \rightarrow \text{ sexpr.} \cdot \text{id} & \text{shift + goto 18} \\
& '\text{def} \rightarrow = \text{sexpr.} & \text{reduce if \{ comma, semicolon \}} \\
& '\text{expr} \rightarrow = \text{sexpr.} & \text{reduce if \{ end, semicolon \}}
\end{align*}
\]
Example 4.7 illustrates the performance of the extended parser on the string of example 4.6 with the conflict in state 56 resolved as discussed above. Notice that the parser now determines a valid interpretation of the string. This same technique can be applied to states 38, 52, and 53.

In some cases, a reduction conflict may occur between a suffix error production and a valid production. This may happen either because some right side in the original grammar is a proper suffix of some other right side, or because two or more suffix productions create an ambiguous structural interpretation of some phrase. For example, item set I_{57} of Appendix II (shown below) contains such a reduction conflict.

\[
I_{57}: \text{stmt-list} \rightarrow \text{stmt}.
\]

reduce if \{ end, semicolon \}

\[\text{stmt-list} \rightarrow \text{semicolon \ stmt. reduce if \{ end, semicolon \}}\]

This conflict is caused by the two conflicting derivations

\[\text{corral} \Rightarrow \text{semicolon \ stmt-list} \Rightarrow \text{semicolon \ stmt}\]

\[\text{stmt-list} \Rightarrow \text{semicolon \ stmt}\]

either of which is a reasonable interpretation given no left context. An arbitrary choice can be made to resolve this conflict, although a decision in favor of the valid production may be preferrable in order to delay the suffix reduction, thus forcing the suffix production to occur as high up in the derivation tree as possible. Use of such heuristic decisions at parser construction time can probably result in almost all inadequate states being resolved without allowing spurious local errors to be detected. For example,
Example 4.7

The reduce/reduce conflict in item set $I_{56}$ is resolved in favor of 'def + = sexpr, since then whatever legally follows a semicolon will be accepted as part of the valid fragment. The parse starts from step 11; steps 1 through 10 are the same as in Example 4.6.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>11. ? 0 = 36 sexpr 56</td>
<td>reduce 'def + = sexpr</td>
</tr>
<tr>
<td>12. ? 0 'def 48</td>
<td>reduce 'def-list + 'def</td>
</tr>
<tr>
<td>13. ? 0 'def-list 47</td>
<td>reduce 'decl + 'def-list</td>
</tr>
<tr>
<td>14. ? 0 'decl 45</td>
<td>reduce 'decl-list + 'decl</td>
</tr>
<tr>
<td>15. ? 0 'decl-list 44</td>
<td>shift</td>
</tr>
<tr>
<td>16. ? 0 'decl-list 44 ; 59</td>
<td>shift</td>
</tr>
<tr>
<td>17. ? 0 'decl-list 44 ; 59 id 4</td>
<td>shift</td>
</tr>
<tr>
<td>18. ? 0 'decl-list 44 ; 59 id 4 + 9</td>
<td>shift</td>
</tr>
<tr>
<td>19. ? 0 'decl-list 44 ; 59 id 4 + 9 id 31</td>
<td>reduce sexpr + id</td>
</tr>
<tr>
<td>20. ? 0 'decl-list 44 ; 59 id 4 + 9 sexpr 14</td>
<td>reduce expr + sexpr</td>
</tr>
<tr>
<td>21. ? 0 'decl-list 44 ; 59 id 4 + 9 expr 30</td>
<td>reduce stmt + id + expr</td>
</tr>
<tr>
<td>22. ? 0 'decl-list 44 ; 59 stmt 25</td>
<td>reduce stmt-list + stmt</td>
</tr>
<tr>
<td>23. ? 0 'decl-list 44 ; 59 stmt-list 64</td>
<td>reduce 'corral + 'decl-list ; stmt-list</td>
</tr>
<tr>
<td>24. ? 0 'corral 43</td>
<td>shift</td>
</tr>
<tr>
<td>25. ? 0 'corral 43 end 68</td>
<td>reduce 'block + 'corral end</td>
</tr>
<tr>
<td>26. ? 0 'block 41</td>
<td>shift</td>
</tr>
<tr>
<td>27. ? 0 'block 41 -</td>
<td>78</td>
</tr>
</tbody>
</table>
Example 4.7 (continued)

This parse determines a valid interpretation of the input string, and leaves the stack containing the condensed string:

\[- \text{begin id} \uparrow \text{'block -'} \]
all but one of the ambiguous transitions (the shift/reduce conflict in state 38) in the parser of Appendix II can be eliminated in this way. However, there may in general be conflicts which cannot be resolved heuristically without introducing the potential for detecting spurious local errors (such as shift/reduce conflicts).

A straightforward approach to handling this problem is suggested by the methods of backing up an "error point" described by Modry [49] and Pennello and DeRemer [52]. Since this nondeterminism is caused by the left-to-right manner of extended parsing using limited right context, a right-to-left pass over the condensed version of the input string in the stack may be able to use the extended condensed context available to the right of discontinuities caused by any remaining inadequate states to either determine that a local error actually exists there, or else eliminate the discontinuity by connecting the two adjacent valid fragments. This approach is rather ad hoc and will not be considered further here. A more intriguing approach is to attempt to eliminate the remaining inadequate states in the finite control by generating new "special purpose" productions which essentially provide arbitrary lookahead (including non-terminals) to "postpone" conflicts, hopefully far enough so that condensed right context can resolve the conflict or so that heuristic resolution is possible.
The general idea behind conflict postponement is similar to Earley's parser compilation algorithms [14], and suffers the same problems of nontermination under certain conditions. The method is to generate new productions derived from the current set of productions in the extended grammar by replacing some nonterminal A where it occurs in a right side by the right side of some A-production. Initial LR(k) items created by these new productions are only added to the restart state. The decision as to what new productions to generate is determined by a particular set of conflicting actions for a particular lookahead symbol in a particular recovery state. The left sides involved in the reduce actions determine which nonterminals need to be replaced, and the corresponding right sides determine the replacements. The particular productions in which a selected nonterminal A is to be replaced are those in which A occurs followed immediately by the lookahead symbol causing the conflict or in which A is the last symbol and the lookahead symbol is a follower of the left side. The technique is not formalized here, but is illustrated by Example 4.8 in which the shift/reduce conflict of item set $I_{38}$ of Appendix II is postponed until it becomes a reduce/reduce conflict which can be resolved heuristically. Example 4.9 illustrates a parse with the parser of Appendix II and with the parser as modified by Example 4.8.
Example 4.8

The shift/reduce conflict with lookahead symbol = in $I_{38}$ of Appendix II can be postponed, converting it into a reduce/reduce conflict by adding the new production

$$expr + id = sexpr$$

to the extended grammar for use only in the restart item set. This production is derived from the two productions

$$expr + sexpr = sexpr$$
$$sexpr + id$$

by replacing $sexpr$ with $id$. The new version of $I_{38}$, along with the new successor states, is shown below, where the reduce action on lookahead symbol = of $sexpr + id$ is replaced by a shift on the new item.

$I_{38}$

<table>
<thead>
<tr>
<th>stmt + id .+ expr</th>
<th>shift + goto 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>def + id.</td>
<td>reduce if { comma, semicolon }</td>
</tr>
<tr>
<td>def + id.= sexpr</td>
<td>shift = goto 79</td>
</tr>
<tr>
<td>sexpr + id.</td>
<td>reduce if { +, end, comma, semicolon }</td>
</tr>
<tr>
<td>expr + id.=sexpr</td>
<td>shift = goto 79</td>
</tr>
</tbody>
</table>
Example 4.8 (continued)

\[ I_{79} \text{ expr } \rightarrow \text{id} = .\text{sexpr} \quad \text{goto} \ 80 \]
\[ \text{def } \rightarrow \text{id} = .\text{sexpr} \quad \text{goto} \ 80 \]
\[ \text{sexpr } \rightarrow .\text{id} \quad \text{shift} \ \text{id} \ \text{goto} \ 31 \]
\[ \text{sexpr } \rightarrow .\text{sexpr} + \text{id} \quad \text{goto} \ 80 \]

\[ I_{80} \text{ expr } \rightarrow \text{id} = \text{sexpr}. \quad \text{reduce} \ \text{if} \ \{ \ \text{end}, \ \text{semicolon} \ \} \]
\[ \text{def } \rightarrow \text{id} = \text{sexpr}. \quad \text{reduce} \ \text{if} \ \{ \ \text{comma}, \ \text{semicolon} \ \} \]
\[ \text{sexpr } \rightarrow \text{sexpr} + \text{id} \quad \text{shift} + \ \text{goto} \ 18 \]

Thus, rather than halt on lookahead symbol = in state 38, the parser can now make at least two additional shift actions on any locally correct input string of length > 2 starting with an =. Furthermore, the reduce/reduce conflict in \( I_{80} \) can be resolved heuristically in favor of the reduction to \text{def}. 
Example 4.9

The parsing process using the parser of Appendix II, modified as indicated in Example 4.8 (with heuristic resolutions noted), is illustrated on the string

\[- id = id ; type - |\]

For comparison, the parse using the unmodified parser is shown first.

<table>
<thead>
<tr>
<th>stack</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $ 0</td>
<td>shift</td>
</tr>
<tr>
<td>2. $ 0</td>
<td>error/restart</td>
</tr>
<tr>
<td>3. $ 0</td>
<td>shift</td>
</tr>
<tr>
<td>4. $ 0</td>
<td>restart¹</td>
</tr>
<tr>
<td>5. # 0</td>
<td>shift</td>
</tr>
<tr>
<td>6. # 0 = 36</td>
<td>shift</td>
</tr>
<tr>
<td>7. # 0 = 36</td>
<td>reduce sexpr → id</td>
</tr>
<tr>
<td>8. # 0 = 36</td>
<td>reduce 'def + = sexpr²</td>
</tr>
<tr>
<td>9. # 0 'def 48</td>
<td>reduce 'def-list → 'def</td>
</tr>
<tr>
<td>10. # 0 'def-list 47</td>
<td>reduce 'decl + 'def-list</td>
</tr>
<tr>
<td>11. # 0 'decl 45</td>
<td>reduce 'decl-list → 'decl</td>
</tr>
<tr>
<td>12. # 0 'decl-list 44</td>
<td>shift</td>
</tr>
<tr>
<td>13. # 0 'decl-list 44 ; 59</td>
<td>shift</td>
</tr>
<tr>
<td>14. # 0 'decl-list 44 ; 59</td>
<td>error/restart</td>
</tr>
<tr>
<td>15. # 0 'decl-list 44 ; 59</td>
<td>shift</td>
</tr>
<tr>
<td>16. # 0 'decl-list 44 ; 59</td>
<td>error accept</td>
</tr>
</tbody>
</table>
Example 4.9 (continued)

Notes

1 - shift/reduce conflict on =

2 - reduce conflict heuristically resolved

The modified parser's actions appear next

<table>
<thead>
<tr>
<th>stack</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $ 0</td>
<td>shift</td>
</tr>
<tr>
<td>2. $ 0</td>
<td>- 1</td>
</tr>
<tr>
<td>3. $ 0</td>
<td>- 1 ? 0</td>
</tr>
<tr>
<td>4. $ 0</td>
<td>- 1 ? 0</td>
</tr>
<tr>
<td>5. $ 0</td>
<td>- 1 ? 0</td>
</tr>
<tr>
<td>6. $ 0</td>
<td>- 1 ? 0</td>
</tr>
<tr>
<td>7. $ 0</td>
<td>- 1 ? 0</td>
</tr>
<tr>
<td>8. $ 0</td>
<td>- 1 ? 0</td>
</tr>
<tr>
<td>9. $ 0</td>
<td>- 1 ? 0</td>
</tr>
<tr>
<td>10. $ 0</td>
<td>- 1 ? 0</td>
</tr>
<tr>
<td>11. $ 0</td>
<td>- 1 ? 0</td>
</tr>
<tr>
<td>12. $ 0</td>
<td>- 1 ? 0</td>
</tr>
<tr>
<td>13. $ 0</td>
<td>- 1 ? 0</td>
</tr>
<tr>
<td>14. ? 0</td>
<td>'decl-list 44 ; 59</td>
</tr>
<tr>
<td>15. ? 0</td>
<td>'decl-list 44 ; 59</td>
</tr>
</tbody>
</table>

Notes

1 - reduce conflict heuristically resolved

Notice that the second parse determines a valid interpretation of the input string.
Algorithm 4.5 Resolve inadequate states

Input. The extended grammar G' and the collection of sets of extended LR(k) items for G, $\Omega_k$, produced by Algorithm 4.3.

Output. $\Omega_k$ with some or all conflicts resolved.

Method.

(1) Identify the recovery item sets of $\Omega_k$ which contain conflicting actions; if there are none then halt.

(2) Resolve actions in each such item set which can be decided heuristically by eliminating items from the set which conflict with the chosen action.

(3) If any conflicting actions cannot be resolved heuristically for some item sets, determine a set of new productions to postpone each conflict and add initial items to the restart state determined by these new productions.

(4) Propagate the new items along paths in $\Omega_k$, replacing the items superceded, until the conflict set is reached.

(5) Compute any new successor item sets.

(6) If the termination condition is satisfied, then halt, otherwise go to step (1).

//
Under certain conditions, this technique could add an infinite number of new productions while attempting to postpone a conflict to a point where heuristic resolution is possible. However, in many cases only a few extra productions are required to eliminate an ambiguous action in a state or postpone it to a point where an arbitrary decision is feasible. In order to implement this within an automatic parser generator, then, some fixed bound must be placed on the resolution phase, or the implementor must be required to set a bound. The bound could be as simple as a time limit, or could be phrased in terms of the number of states added, a maximum length on the right sides of new productions, etc. An automatic resolution algorithm would then proceed as shown in Algorithm 4.5.

If all inadequate states can be resolved, then an extended LR(k) parser determines an extended parse of any input string with unique endmarkers. The maximal valid fragment corresponding to each locally correct string determined by the restarts performed can be recovered from the segmented parse list by expanding all nonterminals reduced by suffix productions. Suffix productions essentially allow the extended LR(k) parser to simulate a noncanonical extended parser. If some inadequate states cannot be resolved, then the extended LR(k) parser determines an approximation of an extended parse, where some of the restarts may not be caused by local errors.
4.6 Optimizations

There are two places where space optimizations can improve the practicality of an extended LR(k) parser. One is during the parsing process, where the stack may tend to become quite large; the other is the space required to store the parser itself. Since an extended LR(k) parser is constructed using the standard LR(k) construction techniques, the space optimizations which are applicable to the LR(k) parsers are also applicable in this case (see for example [3,10,11,30]).

Up to now, no mention has been made of the prefix and infix error productions used in Chapter 3. These productions enabled the general extended parser to condense the input string to a sequence of nonterminals, and can provide the same capability in an extended LR(k) parser. By use of default reductions to be taken on an error parsing action, any valid fragment can be reduced to a single nonterminal using prefix, infix and valid productions. The choice of which default reduction to take in a particular state may require some heuristic decision process, but since the only goal is to condense the stack during the parsing phase, it should not require complicated heuristics. Notice that this directly corresponds to the backward condensation phase of Graham and Rhodes [21] or Druseikis and Ripley [13]. The maximal valid fragments can still be recovered when needed for reparation from the segmented parse list.
Chapter 5
SUMMARY AND CONCLUSIONS

5.1 Summary

In this dissertation, an approach to solving the problem of automatic syntax error analysis is proposed in which the repair (and diagnosis) of syntax errors is separated from the error detection process. This decomposition allows the two subproblems to be treated independently, which is a radical departure from previous approaches to the problem. The goal of the research reported is to develop a syntax error detection process which is defined in terms of the grammar defining a programming language only. Without benefit of knowledge of the global structure of a particular incorrect program and semantic or heuristic information, the actual syntax errors made by a programmer are difficult to determine. However, a tractable model syntactically incorrect programs is proposed which only considers deletion errors within strings, which are assumed to be very nearly correct globally, although local intervals may be structurally corrupted. These constraints on the syntax error inducing process mean that structural features of the language are preserved by the process. Hence, the partial phrase structure of an incorrect program can be used to isolate the intervals within the string where the actual syntax errors are manifested in terms of deletion errors; these syntax error indicators are referred to as
local errors.

A formal model of detection of local errors is developed which extends the language recognition process from sentence recognition to include recognition of sequences of sentence fragments, referred to as locally correct strings. A nondeterministic approach to the implementation of this error detection process is proposed which makes use of the standard machinery of context-free language generation. The original grammar defining the language is extended so that a sequence of sentence fragments of the original grammar is generated by automatically augmenting the grammar with a set of "error" productions which define a sequence of phrases and prefix, suffix or infix phrases of the original grammar. An algorithm which performs error detection according to the model is then presented which essentially parses an arbitrary program with unique end markers according to this extended grammar using the technique of dynamic programming. The algorithm operates in $O(n^3)$ time and $O(n^2)$ space for an input string of length $n$. In addition to error detection, the algorithm can determine a parse of any correct string, and for incorrect strings can determine a partial phrase structure of each locally correct interval of the string, (which may not be unique).
The insight gained from the development of this general algorithm, referred to as extended parsing, is then applied to the development of an efficient deterministic extended parsing algorithm, building upon the work of Druseikis and Ripley [13]. The basic approach taken is to define, during parser construction, a finite set of orthogonal error interpretations which are applied after an error is detected to allow the parser to continue to process the input string deterministically. The parser is essentially oriented toward the recognition of sentence suffixes from its initial state, and each time an error is detected the algorithm restarts in the initial state. This results in a efficient error detecting parser which is "tailor-made" for the specific grammar, as opposed to a "bolt-on" general-purpose recovery algorithm which externally augments a parser achieving an ad hoc form of error detection.

A class of error detecting parsers is defined which is a generalization of the LR(k) class of parsers, referred to as the extended LR(k) parsers. An extended LR(k) parser is constructed for a specific grammar in a manner similar to the method by which an LR(k) parser is constructed for the grammar, simulating at parser construction time, in a limited sense, the possible local error interpretations that can occur during parsing. In all cases, an extended LR(k) parsing automaton contains an LR(k) parsing automaton of the same variety in a subset of its finite control, so that correct inputs are
processed as if by an LR(k) parser and thus do not suffer
form the extended capabilities, other than as due to the
increased size of the parser's finite control. Because the
extended LR(k) parsing algorithm is the usual LR(k) parsing
algorithm with a standard next move defined after detecting
an error, an arbitrary input string with unique end markers
is processed in linear time and space regardless of the amount
of corruption of the string.

A straightforward application of techniques used in the
general extended parsing algorithm to the construction of an
extended LR(k) parser results in an automaton which may enter
configurations which are not deterministic. Heuristic
techniques are therefore proposed to constrain the algorithm
to produce only one parse of any sentence fragment by select-
ing one interpretation from among a set of conflicting ones
where possible and by restructuring the finite control to
specialize the recognition of certain phrase fragment pre-
fixes in an attempt to delay conflicts until extended right
context either disambiguates the interpretation or allows an
arbitrary selection as above. This latter technique is
accomplished by generating entirely new productions which are
only used where required to delay conflicts.
5.2 Limitations

The model of error detection proposed in this dissertation has a few limitations. A valid interpretation of an incorrect string is not necessarily unique, although it may be unique in many cases. That is, the location of a local error may not be unique. Furthermore, a local parse of an interval determined by a valid interpretation may not be entirely unique, depending on the length of the fragment and the amount of redundant language features contained within the fragment. This means that an extended parser may not be able to determine the globally correct structure of a locally correct substring. While this is not a fatal objection, it does require that the error reparation strategy be prepared to cope with this eventuality.

An inherent restriction imposed by this approach is that semantic analysis methods which operate in parallel with a parser and depend on a canonical reduction sequence cannot be used after an error is detected, since extended parsing is an intrinsically noncanonical process. In addition, the presence of ambiguous transitions in a practical extended parser limits the feasibility of performing any semantic analysis on an incorrect string during the parsing phase, since the techniques proposed for eliminating this non-determinism may result in an incorrect interpretation of part of the string, so that local semantic analysis may have to be undone. This suggests that semantic analysis should properly be separated completely from the syntactic
analysis phase if such analysis is desired on incorrect strings. Furthermore, this would allow a noncanonical base parser to be used, such as Fraley's parse ahead algorithm [18], which could potentially eliminate many cases of ambiguous transitions.
5.3 Significance of this work

The model of syntax error detection developed in this dissertation does not require recovery or reparation to locate errors in an incorrect string, and as such is the first of its kind. It is also the only formal model of error detection which can be implemented directly and which can also be implemented with a practical algorithm. This model has the desirable property that the definition of an error in an incorrect string depends only on the grammar used to define the language, so that the error detection process is independent of the repairs required to transform the string into a sentence; the only other class of formal models (minimum distance reparation) defines errors a posterior by the repairs made, and is sensitive to the distance metric used. Error detection with respect to the model works somewhat like a human would, by identifying locally correct intervals of the incorrect string to isolate syntax errors, making maximum use of every symbol within the string. It is also straightforward to identify the partial phrase structure of each locally correct interval. The direct result of the development of a general algorithm to implement the model is the identification of practical recovery-based approaches to error detection as an integral part of the parsing process rather than by using an external recovery mechanism, so that any input string is processed uniformly by the same mechanism.
The approach to extend LR(k) parsing yields immediate improvements to previous work in error recovery by forward context condensation. Previous methods of context condensation as applied to simple precedence and LR(k) parsers work by reducing (maximal) valid fragments of right sentential forms, which are referred to as (maximal) derived valid fragments in [48], whereas the use of phrase suffix reductions can simulate a noncanonical parsing process permitting valid fragments of arbitrary sentential forms to be reduced, which allows longer deterministic forward moves to occur. The LR(k) context condensation algorithm of Druseikis and Ripley [13] is made more efficient by the use of phrase suffix reductions, since the relative position of the most recent error is then available in the finite control of the parser rather than only in the stack. This eliminates the need for the parser to examine the stack contents popped during a reduction. Furthermore, the construction technique employed in this dissertation to compute an extended LR(k) parser generalizes the method of Druseikis and Ripley for application to any LR(k)-like parser, and identifies a method of approaching the size computation for these parsers.

The real significance of this approach to syntax error analysis is that it provides a uniform basis from which to investigate automatic error reparation and diagnosis independently from the error detection mechanism. Within this framework, error repair need only depend on knowledge of a
language and its grammar, and requires no unnecessary interactions with the error detection process, thus providing a clean, modular structure for the syntax analysis phase of a compiler. The error reparation process can make use of information deduced from an external parse, such as distribution and density of local errors or length and structure of locally correct segments to direct its analysis of the incorrect string and to determine a globally consistent repair of that string.
5.4 Recommendations for further research

Although the practical extended parsing method presented in this dissertation is based on LR(k) parsers, a similar approach with other bottom-up and partially bottom-up parsing techniques, such as the various types of precedence parsers and recursive finite automata [37,59], appears feasible. With extensions to the method of generating error productions to handle general regular expressions, this approach to automatic error detection could be incorporated into parsers for regular right part grammars [33,39] as well. It is not immediately apparent, however, whether this approach is feasible with strictly top-down parsing.

An improvement over the extended LR(k) parsing method described here is possible simply by using Fraley's parseahead generalization of LR(k) parsing [18] for the base parser, or some other noncanonical parsing strategy. Other improvements might be achieved through experimentation with heuristics for restricting the number of suffix error productions used in construction of the restart mechanism, or through grammatical restrictions designed to minimize the occurrence of ambiguous recovery transitions. It may prove advantageous to incorporate implementor-supplied error productions in the construction of an extended parser, using heuristic techniques such as proposed here to disambiguate conflicting actions. In addition, language design for ease
of reparation, as noted by Wetherall [60], is an important issue; redundancy in language constructs can help ensure that the locally correct segments of a program have unique structural interpretations and that a local error is isolated within a short error interval.

Development of methods for automatically generating reparation and diagnostic algorithms which can make use of the structural information available in an extended parse of an incorrect program is necessary before completely automatic generation of the syntax analysis phase of a compiler can become a reality. As noted by Birrell [6], an extremely difficult aspect of this phase is the problem of coping with mismatched brackets, because of their global effects on the interpretation of an incorrect program. Global analysis techniques, such as are utilized by Meertens and van Vliet [42,43] and Dieterich [12], applicable to the incorrect extract of a program represented by an extended parse could prove to be useful approaches to this problem. Modifying local error reparation methods for application to an extended parse taken from the extensive collection which have been reported in the literature should also be considered.
APPENDIX I
AN SLR(1) PARSER

The augmented grammar

1. block $\rightarrow$ begin corral end
2. corral $\rightarrow$ decl-list semicolon stmt-list
3. corral $\rightarrow$ stmt-list
4. decl-list $\rightarrow$ decl
5. decl-list $\rightarrow$ decl-list semicolon decl
6. stmt-list $\rightarrow$ stmt
7. stmt-list $\rightarrow$ stmt-list semicolon stmt
8. decl $\rightarrow$ type def-list
9. def-list $\rightarrow$ def
10. def-list $\rightarrow$ def-list comma def
11. def $\rightarrow$ id
12. def $\rightarrow$ id = sexpr
13. stmt $\rightarrow$ block
14. stmt $\rightarrow$ id + expr
15. expr $\rightarrow$ sexpr
16. expr $\rightarrow$ sexpr = sexpr
17. sexpr $\rightarrow$ id
18. sexpr $\rightarrow$ sexpr + id
19. sentence $\rightarrow$ |- block -|
FOLLOW (block) = \{ -|, end, semicolon \}
FOLLOW (corral) = \{ end \}
FOLLOW (decl-list) = \{ semicolon \}
FOLLOW (stmt-list) = \{ end, semicolon \}
FOLLOW (decl) = \{ semicolon \}
FOLLOW (def-list) = \{ comma, semicolon \}
FOLLOW (def) = \{ comma, semicolon \}
FOLLOW (stmt) = \{ end, semicolon \}
FOLLOW (expr) = \{ end, semicolon \}
FOLLOW (sexpr) = \{ +, =, end, comma, semicolon \}
FOLLOW (sentence) = \epsilon

SLR(1) item sets

I₀ sentence → .- block -| shift |- goto 1

I₁ sentence → |-.block -| goto 3
   block .begin corral end shift begin goto 2
I₂  block → \text{begin\_corral\_en} \quad \text{goto 6}
    corral → \text{.decl-list \_semicolon\_stmt-list} \quad \text{goto 7}
    corral → \text{.stmt-list} \quad \text{goto 8}
    decl-list → \text{.decl} \quad \text{goto 23}
    decl-list → \text{.decl-list \_semicolon\_decl} \quad \text{goto 7}
    stmt-list → \text{.stmt} \quad \text{goto 25}
    stmt-list → \text{.stmt-list \_semicolon\_stmt} \quad \text{goto 8}
    decl → \text{.type\_def-list} \quad \text{shift type goto 5}
    stmt → \text{.block} \quad \text{goto 29}
    stmt → \text{.id\_+\_expr} \quad \text{shift id goto 4}
    block → \text{.begin\_corral\_end} \quad \text{shift begin goto 2}

I₃  sentence → \mid - \text{block.-} \mid \quad \text{shift -| goto 33}

I₄  stmt → \text{id\_.+\_expr} \quad \text{shift + goto 9}

I₅  decl → \text{.type\_def-list} \quad \text{goto 11}
    def-list → \text{.def} \quad \text{goto 27}
    def-list → \text{.def-list \_comma\_def} \quad \text{goto 11}
    def → \text{.id} \quad \text{shift id goto 10}
    def → \text{.id\_=\_sexpr} \quad \text{shift id goto 10}

I₆  block → \text{begin\_corral\_end} \quad \text{shift end goto 22}
\[ I_7 \quad \text{corral} \rightarrow \text{decl-list}.\text{semicolon} \text{stmt-list} \quad \text{shift semicolon goto 12} \]
\[ \text{decl-list} \rightarrow \text{decl-list}.\text{semicolon} \text{decl} \quad \text{shift semicolon goto 12} \]
\[ I_8 \quad \text{corral} \rightarrow \text{stmt-list.} \quad \text{reduce if \{ end \}} \]
\[ \text{stmt-list} \rightarrow \text{stmt-list}.\text{semicolon} \text{stmt} \quad \text{shift semicolon goto 13} \]
\[ I_9 \quad \text{stmt} \rightarrow \text{id}+.\text{expr} \quad \text{goto 30} \]
\[ \text{expr} \rightarrow .\text{sexpr} \quad \text{goto 14} \]
\[ \text{expr} \rightarrow .\text{sexpr} = \text{sexpr} \quad \text{goto 14} \]
\[ \text{sexpr} \rightarrow .\text{id} \quad \text{shift id goto 31} \]
\[ \text{sexpr} \rightarrow .\text{sexpr} + \text{id} \quad \text{goto 14} \]
\[ I_{10} \quad \text{def} \rightarrow \text{id.} \quad \text{reduce if \{ comma, semicolon \}} \]
\[ \text{def} \rightarrow \text{id}+.\text{sexpr} \quad \text{shift = goto 15} \]
\[ I_{11} \quad \text{decl} \rightarrow \text{type} \text{def-list.} \quad \text{reduce if \{ semicolon \}} \]
\[ \text{def-list} \rightarrow \text{def-list}.\text{comma} \text{def} \quad \text{shift comma goto 16} \]
\[ I_{12} \quad \text{corral} \rightarrow \text{decl-list}.\text{semicolon} \text{stmt-list} \quad \text{goto 17} \]
\[ \text{decl-list} \rightarrow \text{decl-list}.\text{semicolon} \text{decl} \quad \text{goto 24} \]
\[ \text{stmt-list} \rightarrow .\text{stmt} \quad \text{goto 25} \]
\[ \text{stmt-list} \rightarrow \text{stmt-list}.\text{semicolon} \text{stmt} \quad \text{goto 17} \]
\[ \text{decl} \rightarrow \text{type} \text{def-list} \quad \text{shift type goto 5} \]
\[ \text{stmt} \rightarrow .\text{block} \quad \text{goto 29} \]
\[ \text{stmt} \rightarrow \text{id}+.\text{expr} \quad \text{shift id goto 4} \]
\[ \text{block} \rightarrow \text{begin} \text{corral end} \quad \text{shift begin goto 2} \]
I_{13} \quad \text{stmt-list} \rightarrow \text{stmt-list \ semicolon \ stmt} \quad \text{goto 26}

\quad \text{stmt} \rightarrow \cdot \text{block} \quad \text{goto 29}

\quad \text{stmt} \rightarrow \cdot \text{id} + \text{expr} \quad \text{shift id goto 4}

\quad \text{block} \rightarrow \cdot \text{begin \ corral \ end} \quad \text{shift begin goto 2}

I_{14} \quad \text{expr} \rightarrow \text{sexpr.} \quad \text{reduce if \{ end, semicolon \}}

\quad \text{expr} \rightarrow \text{sexpr.=} \text{sexpr} \quad \text{shift = goto 19}

\quad \text{sexpr} \rightarrow \text{sexpr.} + \text{id} \quad \text{shift + goto 18}

I_{15} \quad \text{def} \rightarrow \cdot \text{id} = \text{.sexpr} \quad \text{goto 20}

\quad \text{sexpr} \rightarrow \cdot \text{id} \quad \text{shift id goto 31}

\quad \text{sexpr} \rightarrow \cdot \text{sexpr + id} \quad \text{goto 20}

I_{16} \quad \text{def-list} \rightarrow \text{def-list \ comma \ def} \quad \text{goto 28}

\quad \text{def} \rightarrow \cdot \text{id} \quad \text{shift id goto 10}

\quad \text{def} \rightarrow \cdot \text{id} = \text{sexpr} \quad \text{shift id goto 10}

I_{17} \quad \text{corral} \rightarrow \text{decl-list \ semicolon \ stmt-list.} \quad \text{reduce if \{ end \}}

\quad \text{stmt-list} \rightarrow \text{stmt-list. semicolon \ stmt} \quad \text{shift semicolon goto 13}

I_{18} \quad \text{sexpr} \rightarrow \text{sexpr + id} \quad \text{shift id goto 32}
$I_{19}$  $\text{expr} \rightarrow \text{sexpr} = \text{sexpr}$  
$\text{sexpr} \rightarrow \text{id}$  
$\text{sexpr} \rightarrow \text{sexpr} + \text{id}$  
$\text{goto} \ 21$

$\text{shift \ id \ goto} \ 31$

$\text{goto} \ 21$

$I_{20}$  $\text{def} \rightarrow \text{id} = \text{sexpr}.$  
$\text{sexpr} \rightarrow \text{sexpr} + \text{id}$  
$\text{reduce \ if \ \{ \text{comma, \ semicolon} \}}$

$\text{shift \ + \ goto} \ 18$

$I_{21}$  $\text{expr} \rightarrow \text{sexpr} = \text{sexpr}.$  
$\text{sexpr} \rightarrow \text{sexpr} + \text{id}$  
$\text{reduce \ if \ \{ \text{end, \ semicolon} \}}$

$\text{shift \ + \ goto} \ 18$

$I_{22}$  $\text{block} \rightarrow \text{begin} \ \text{corral} \ \text{end}.$  
$\text{reduce \ if \ \{ \text{-}, \ \text{end, \ semicolon} \}}$

$I_{23}$  $\text{decl-list} \rightarrow \text{decl}.$  
$\text{reduce \ if \ \{ \text{semicolon} \}}$

$I_{24}$  $\text{decl-list} \rightarrow \text{decl-list} \ \text{semicolon} \ \text{decl}.$  
$\text{reduce \ if \ \{ \text{semicolon} \}}$

$I_{25}$  $\text{stmt-list} \rightarrow \text{stmt}.$  
$\text{reduce \ if \ \{ \text{end, \ semicolon} \}}$

$I_{26}$  $\text{stmt-list} \rightarrow \text{stmt-list} \ \text{semicolon} \ \text{stmt}.$  
$\text{reduce \ if \ \{ \text{end, \ semicolon} \}}$

$I_{27}$  $\text{def-list} \rightarrow \text{def}.$  
$\text{reduce \ if \ \{ \text{comma, \ semicolon} \}}$

$I_{28}$  $\text{def-list} \rightarrow \text{def-list} \ \text{comma} \ \text{def}.$  
$\text{reduce \ if \ \{ \text{comma, \ semicolon} \}}$

$I_{29}$  $\text{stmt} \rightarrow \text{block}.$  
$\text{reduce \ if \ \{ \text{end, \ semicolon} \}}$
I_{30} \text{ stmt } \rightarrow \text{id } \leftarrow \text{ expr.} \quad \text{reduce if } \{ \text{end, semicolon}\}

I_{31} \text{ sexpr } \rightarrow \text{id.} \quad \text{reduce if } \{ \text{+, =, end, comma, semicolon}\}

I_{32} \text{ sexpr } \rightarrow \text{sexpr + id.} \quad \text{reduce if } \{ \text{+, =, end, comma, semicolon}\}

I_{33} \text{ sentence } \rightarrow \left| - \text{ block } - \right| \cdot \quad \text{accept}
APPENDIX II
AN EXTENDED SLR(1) PARSER

The suffix error productions added to the grammar of Appendix I

20. 'sentence -> block -
21. 'sentence -> -
22. 'sentence -> 'block -
23. 'block -> corral end
24. 'block -> end
25. 'block -> 'corral end
26. 'corral -> semicolon stmt-list
27. 'corral -> 'stmt-list
28. 'corral -> 'decl-list semicolon stmt-list
29. 'stmt-list -> semicolon stmt
30. 'stmt-list -> 'stmt
31. 'stmt-list -> 'stmt-list semicolon stmt
32. 'decl-list -> semicolon decl
33. 'decl-list -> 'decl
34. 'decl-list -> 'decl-list semicolon decl
35. 'decl -> def-list
36. 'decl -> 'def-list
37. 'stmt -> ++ expr
38. 'stmt -> expr
39. 'stmt -> 'expr
40. 'stmt -> 'block
41. 'def-list → comma def
42. 'def-list → 'def
43. 'def-list → 'def-list comma def
44. 'def → = sexpr
45. 'def → sexpr
46. 'def → 'sexpr
47. 'sexpr → + id
48. 'sexpr → 'sexpr + id
49. 'expr → = sexpr
50. 'expr → 'sexpr
51. 'expr → 'sexpr = sexpr
SLR(1) recovery item sets

\[ I_{79} = I_0 \]

\[
\text{sentence} \rightarrow .|- \text{ block } |- \\
'b \text{sentence} \rightarrow .| \text{ block } |- \\
'b \text{sentence} \rightarrow .|- \\
'b \text{sentence} \rightarrow .' \text{block } |- \\
\text{block} \rightarrow .| \text{begin} \text{ corrall } \text{ end} \\
\text{corrall} \rightarrow .| \text{ decl-list } \text{ semicolon } \text{ stmt-list} \\
\text{corrall} \rightarrow .| \text{ stmt-list} \\
\text{decl-list} \rightarrow .| \text{ decl} \\
\text{decl-list} \rightarrow .| \text{ decl-list } \text{ semicolon } \text{ decl} \\
\text{stmt-list} \rightarrow .| \text{ stmt} \\
\text{stmt-list} \rightarrow .| \text{ stmt-list } \text{ semicolon } \text{ stmt} \\
\text{decl} \rightarrow .| \text{ type } \text{ def-list} \\
\text{stmt} \rightarrow .| \text{ block} \\
\text{stmt} \rightarrow .| \text{ id } \text{ + } \text{ expr} \\
\text{def-list} \rightarrow .| \text{ def} \\
\text{def-list} \rightarrow .| \text{ def-list } \text{ comma } \text{ def} \\
\text{def} \rightarrow .| \text{ id} \\
\text{def} \rightarrow .| \text{ id } = \text{ sexpr} \\
\text{sexpr} \rightarrow .| \text{ id} \\
\text{sexpr} \rightarrow .| \text{ sexpr } + \text{ id} \\
\text{expr} \rightarrow .| \text{ sexpr} \\
\text{expr} \rightarrow .| \text{ sexpr } = \text{ sexpr} \\
'b \text{block} \rightarrow .| \text{ corrall } \text{ end} \\
'b \text{block} \rightarrow .| \text{ end} \\
'b \text{block} \rightarrow .| \text{ corrall } \text{ end}
\]

\[
\text{shift} |- \text{ goto } 1 \\
\text{goto } 40 \\
\text{goto } 77 \\
\text{goto } 41 \\
\text{shift begin } \text{ goto } 2 \\
\text{goto } 7 \\
\text{goto } 8 \\
\text{goto } 23 \\
\text{goto } 7 \\
\text{goto } 25 \\
\text{goto } 8 \\
\text{shift type } \text{ goto } 5 \\
\text{goto } 40 \\
\text{shift id } \text{ goto } 38 \\
\text{goto } 27 \\
\text{goto } 46 \\
\text{shift id } \text{ goto } 38 \\
\text{shift id } \text{ goto } 38 \\
\text{shift id } \text{ goto } 38 \\
\text{goto } 52 \\
\text{goto } 52 \\
\text{goto } 52 \\
\text{goto } 42 \\
\text{shift end } \text{ goto } 67 \\
\text{goto } 43 \]
'corral \rightarrow \text{;semicolontext} \text{stmt-list} \quad \text{shift \; semicolon \; goto \; 39}

'corral \rightarrow \text{'stmt-list} \quad \text{goto \; 54}

'corral \rightarrow \text{'decl-list \; semicolon \; stmt-list} \quad \text{goto \; 44}

'stmt-list \rightarrow \text{;semicolontext} \text{stmt} \quad \text{shift \; semicolon \; goto \; 39}

'stmt-list \rightarrow \text{'stmt} \quad \text{goto \; 55}

'stmt-list \rightarrow \text{'stmt-list \; semicolon \; stmt} \quad \text{goto \; 54}

'decl-list \rightarrow \text{;semicolontext} \text{decl} \quad \text{shift \; semicolon \; goto \; 39}

'decl-list \rightarrow \text{'decl} \quad \text{goto \; 45}

'decl-list \rightarrow \text{'decl-list \; semicolon \; decl} \quad \text{goto \; 44}

'decl \rightarrow \text{def-list} \quad \text{goto \; 46}

'decl \rightarrow \text{'def-list} \quad \text{goto \; 47}

'stmt \rightarrow \text{+ \; expr} \quad \text{shift \; + \; goto \; 35}

'stmt \rightarrow \text{expr} \quad \text{goto \; 49}

'stmt \rightarrow \text{'expr} \quad \text{goto \; 50}

'stmt \rightarrow \text{'block} \quad \text{goto \; 41}

'def-list \rightarrow \text{,comma \; def} \quad \text{shift \; comma \; goto \; 37}

'def-list \rightarrow \text{'def} \quad \text{goto \; 48}

'def-list \rightarrow \text{'def-list \; comma \; def} \quad \text{goto \; 47}

'def \rightarrow \text{= \; sexpr} \quad \text{shift \; = \; goto \; 36}

'def \rightarrow \text{sexpr} \quad \text{goto \; 52}

'def \rightarrow \text{'sexpr} \quad \text{goto \; 53}

'sexpr \rightarrow \text{+ \; id} \quad \text{shift \; + \; goto \; 34}

'sexpr \rightarrow \text{'sexpr \; + \; id} \quad \text{goto \; 53}

'expr \rightarrow \text{= \; sexpr} \quad \text{shift \; = \; goto \; 36}

'expr \rightarrow \text{'sexpr} \quad \text{goto \; 53}

'expr \rightarrow \text{'sexpr \; = \; sexpr} \quad \text{goto \; 53}
\[ I_{34} \quad \text{sexpr} \rightarrow +.id \quad \text{shift id goto 75} \]

\[ I_{35} \quad \text{stmt} \rightarrow +.expr \quad \text{goto 74} \]
\[ \text{expr} \rightarrow .sexpr \quad \text{goto 14} \]
\[ \text{expr} \rightarrow .sexpr = \text{sexpr} \quad \text{goto 14} \]
\[ \text{sexpr} \rightarrow +.id \quad \text{shift id goto 31} \]
\[ \text{sexpr} \rightarrow \text{sexpr + id} \quad \text{goto 14} \]

\[ I_{36} \quad \text{def} \rightarrow =.sexpr \quad \text{goto 56} \]
\[ \text{expr} \rightarrow =.sexpr \quad \text{goto 56} \]
\[ \text{sexpr} \rightarrow +.id \quad \text{shift id goto 31} \]
\[ \text{sexpr} \rightarrow \text{sexpr + id} \quad \text{goto 56} \]

\[ I_{37} \quad \text{def-list} \rightarrow \text{comma.def} \quad \text{goto 72} \]
\[ \text{def} \rightarrow .id \quad \text{shift id goto 10} \]
\[ \text{def} \rightarrow .id = \text{sexpr} \quad \text{shift id goto 10} \]

\[ I_{38} \quad \text{stmt} \rightarrow \text{id}-.expr \quad \text{shift + goto 9} \]
\[ \text{def} \rightarrow \text{id} \quad \text{reduce if \{ comma, semicolon \}} \]
\[ \text{def} \rightarrow \text{id} = \text{sexpr} \quad \text{shift = goto 15} \]
\[ \text{sexpr} \rightarrow \text{id} \quad \text{reduce if \{ +, =, end, comma, semicolon \}} \]

Note: shift/reduce conflict on \{ = \}
reduce/reduce conflict on \{ comma, semicolon \}
I₃⁹ 'corral → `semicolon stmt-list
go to 58
'sstmt-list → `semicolon stmt
go to 57
'decl-list → `semicolon decl
go to 69
stmt-list → .stmt
go to 57
stmt-list → .stmt-list `semicolon stmt
go to 58
stmt → .block
go to 29
stmt → .id + expr
shift id goto 4
decl → .type def-list
shift type goto 5
block → .begin corral end
shift begin goto 2

I₄₀ stmt → block.
'reduce if { end, semicolon }
'sentence → block.-|
shift -| goto 51

I₄₁ 'stmt → 'block.
'reduce if { end, semicolon }
'sentence → 'block.-|
shift -| goto 78

I₄₂ 'block → corral.end
shift end goto 66

I₄₃ 'block → 'corral.end
shift end goto 68

I₄₄ 'corral → 'decl-list `semicolon stmt-list
shift semicolon goto 59
'decl-list → 'decl-list `semicolon decl
shift semicolon goto 59

I₄₅ 'decl-list → 'decl.
'reduce if { semicolon }
I_{46} \quad \text{def-list} \rightarrow \text{def-list.comma def} \quad \text{shift comma goto 16}
\quad \text{'decl} \rightarrow \text{def-list.}
\quad \text{reduce if \{ \text{semicolon} \}}

I_{47} \quad \text{'decl} \rightarrow \text{'def-list.}
\quad \text{reduce if \{ \text{semicolon} \}}
\quad \text{shift comma goto 60}
\quad \text{'def-list} \rightarrow \text{'def-list.comma def}

I_{48} \quad \text{'def-list} \rightarrow \text{'def.}
\quad \text{reduce if \{ comma, semicolon \}}

I_{49} \quad \text{'stmt} \rightarrow \text{expr.}
\quad \text{reduce if \{ end, semicolon \}}

I_{50} \quad \text{'stmt} \rightarrow \text{'expr.}
\quad \text{reduce if \{ end, semicolon \}}

I_{51} \quad \text{'sentence} \rightarrow \text{block |.}
\quad \text{error accept}

I_{52} \quad \text{expr} \rightarrow \text{sexpr.}
\quad \text{reduce if \{ end, semicolon \}}
\quad \text{expr} \rightarrow \text{sexpr=} \text{sexpr}
\quad \text{shift = goto 19}
\quad \text{sexpr} \rightarrow \text{sexpr+ id}
\quad \text{shift + goto 18}
\quad \text{'def} \rightarrow \text{sexpr.}
\quad \text{reduce if \{ comma, semicolon \}}

Note: reduce/reduce conflict on \{ semicolon \}

I_{53} \quad \text{'def} \rightarrow \text{'sexpr.}
\quad \text{reduce if \{ comma, semicolon \}}
\quad \text{'sexpr} \rightarrow \text{'sexpr+ id}
\quad \text{shift + goto 61}
\quad \text{'expr} \rightarrow \text{'sexpr.}
\quad \text{reduce if \{ end, semicolon \}}
\quad \text{'expr} \rightarrow \text{'sexpr=} \text{sexpr}
\quad \text{shift = goto 62}

Note: reduce/reduce conflict on \{ semicolon \}
I₅₄  'corral → 'stmt-list.
  reduce if { end }
  'stmt-list → 'stmt-list.semicolon.stmt  shift semicolon goto 63

I₅₅  'stmt-list → 'stmt.
  reduce if { end, semicolon }

I₅₆  sexpr → sexpr.+ id
    'def → = sexpr.
    'expr → = sexpr.
  reduce if { comma, semicolon }
  reduce if { end, semicolon }

Note: reduce/reduce conflict on { semicolon }

I₅₇  stmt-list → stmt.
  reduce if { end, semicolon }
  'stmt-list → semicolon stmt.
  reduce if { end, semicolon }

Note: reduce/reduce conflict on { end, semicolon }

I₅₈  stmt-list → stmt-list.semicolon stmt
  shift semicolon goto 13
  'corral → semicolon stmt-list.
  reduce if { end }

I₅₉  'corral → 'decl-list semicolon stmt-list goto 64
  'decl-list → 'decl-list semicolon.decl goto 70
  stmt-list → .stmt goto 25
  stmt-list → .stmt-list semicolon stmt goto 64
  decl → .type def-list
  shift type goto 5
  stmt → .block goto 29
  stmt → .id + expr
  shift id goto 4
  block + .begin corral end
  shift begin goto 2
I_{60} \quad \text{def-list} \to \text{def-list comma.def} \quad \text{goto 73}
  \quad \text{def} \to .id \quad \text{shift id goto 10}
  \quad \text{def} \to .id = \text{sexpr} \quad \text{shift id goto 10}

I_{61} \quad \text{sexpr} \to \text{sexpr + .id} \quad \text{shift id goto 76}

I_{62} \quad \text{expr} \to \text{sexpr = .sexpr} \quad \text{goto 65}
  \quad \text{sexpr} \to .id \quad \text{shift id goto 31}
  \quad \text{sexpr} \to \text{sexpr + id} \quad \text{goto 65}

I_{63} \quad \text{stmt-list} \to \text{stmt-list semicolon stmt} \quad \text{goto 71}
  \quad \text{stmt} \to .block \quad \text{goto 29}
  \quad \text{stmt} \to .id + \text{expr} \quad \text{shift id goto 4}
  \quad \text{block} \to .\text{begin corral end} \quad \text{shift begin goto 2}

I_{64} \quad \text{corral} \to \text{decl-list semicolon stmt-list.} \text{ reduce if \{ end \}}
  \quad \text{stmt-list} \to \text{stmt-list semicolon stmt} \quad \text{shift semicolon goto 13}

I_{65} \quad \text{expr} \to \text{sexpr = sexpr.} \quad \text{reduce if \{ end, semicolon \}}
  \quad \text{sexpr} \to \text{sexpr + id} \quad \text{shift + goto 18}

I_{66} \quad \text{block} \to \text{corral end.} \quad \text{reduce if \{-, end, semicolon \}}

I_{67} \quad \text{block} \to \text{end.} \quad \text{reduce if \{-, end, semicolon \}}

I_{68} \quad \text{block} \to \text{corral end.} \quad \text{reduce if \{-, end, semicolon \}}
\[ I_{69} \quad 'decl\text{-}list \rightarrow \text{semicolon } \text{decl.} \quad \text{reduce if } \{ \text{semicolon} \} \]

\[ I_{70} \quad 'decl\text{-}list \rightarrow 'decl\text{-}list \text{semicolon } \text{decl.} \quad \text{reduce if } \{ \text{semicolon} \} \]

\[ I_{71} \quad 'stmt\text{-}list \rightarrow 'stmt\text{-}list \text{semicolon } \text{stmt.} \quad \text{reduce if } \{ \text{end, semicolon} \} \]

\[ I_{72} \quad 'def\text{-}list \rightarrow \text{comma } \text{def.} \quad \text{reduce if } \{ \text{comma, semicolon} \} \]

\[ I_{73} \quad 'def\text{-}list \rightarrow 'def\text{-}list \text{comma } \text{def.} \quad \text{reduce if } \{ \text{comma, semicolon} \} \]

\[ I_{74} \quad 'stmt \rightarrow \text{=} \text{expr.} \quad \text{reduce if } \{ \text{end, semicolon} \} \]

\[ I_{75} \quad 'sexpr \rightarrow \text{+ } \text{id.} \quad \text{reduce if } \{ \text{+, =, end, comma, semicolon} \} \]

\[ I_{76} \quad 'sexpr \rightarrow 'sexpr \text{+ } \text{id.} \quad \text{reduce if } \{ \text{+, =, end, comma, semicolon} \} \]

\[ I_{77} \quad 'sentence \rightarrow \text{-}|. \quad \text{error accept} \]

\[ I_{78} \quad 'sentence \rightarrow 'block \text{-}|. \quad \text{error accept} \]
REFERENCES


[12] Dieterich, E.W., "Parsing and syntactic error recovery for context-free grammars by means of coarse struc-
tures", in Goos, G. and Hartmanis, J. (eds): Automata,
Languages and Programming, Lecture Notes in Computer

simple LR(k) parsers", in Proceedings of the ACM Annual

[14] Earley, J., "An efficient context-free parsing algo-
 rithm", Ph.D. thesis, Department of Computer Science,

[15] Earley, J., "An efficient context-free parsing algo-

of syntax errors", Software - Practice and Experience
6, 2, 1976.

efficient insertion-only error-corrector for LL(1)
parsers", in Conference Record of the Fourth ACM
Symposium on Principles of Programming Languages, Los
Angeles, California, January 1977, p. 97-103.

Technical Report 39, Academic Computing Center,
University of Wisconsin at Madison, June 1975.

a framework for producing compact deterministic
parsers", part I and II, Journal of Computer and System

recovery in compilers", in Conference Record of the ACM
Symposium on Principles of Programming Languages, Boston,
Massachusetts, January 1973, p. 52-58.


[22] Hamming, R.W., "Error-detecting and correcting codes",

[23] Hopcroft, J.E. and Ullman, J.D., "Error correction for
formal languages", Technical Report 52, Department of
Electrical Engineering, Princeton University,
November 1966.


[38] Lyon, G., "Syntax directed least errors analysis for context-free languages - a practical approach", C. ACM 17, 1, January 1974, p. 3-14.


[54] Schwartz, J.T., unpublished SETL notes, Courant Institute of Mathematical Sciences, New York University, undated.


