LOCALLY LEAST-COST ERROR CORRECTORS FOR CONTEXT-FREE AND CONTEXT-SENSITIVE PARSERS

by

Bernard Andre Dion

Computer Sciences Technical Report #344

December 1978

LOCALLY LEAST-COST ERROR CORRECTORS

FOR CONTEXT-FREE AND CONTEXT-SENSITIVE PARSERS†

BY

BERNARD ANDRE DION

A thesis submitted in partial fulfillment of the

requirements of the degree of

Doctor of Philosophy

(Computer Sciences)

at the

University of Wisconsin - Madison

1978

† This research was supported in part by the National Science Foundation under grant number MCS 78-02570

Copyright © by Bernard André Dion

LOCALLY LEAST-COST ERROR CORRECTORS FOR CONTEXT-FREE AND CONTEXT-SENSITIVE PARSERS

by Bernard André Dion

Under the supervision of Assistant Professor Charles N. Fischer

ABSTRACT

A model of error correction is presented. Upon detection of a syntax error, a locally least-cost corrector operates by deleting Ø or more input symbols and inserting a terminal string that guarantees that the first non-deleted symbol will be accepted by the parser. The total correction cost, as defined by a table of deletion and insertion costs, is minimized.

Previous work with the LL(1) parsing technique is summarized and a locally least-cost error corrector for LR(1)-based parsers is developed. Correctness as well as time and space complexity are discussed. In particular, linearity is established in the case of a bounded depth parse stack. Implementation results are presented.

Attributed grammars can be used to specify the context-sensitive syntax of programming languages. A formal presentation of Attribute-Free LL(1) parsing is given and a locally least-cost error corrector for AF-LL(1) parsers is developed for the case in which the attributes that control

context-sensitive correctness have finite domains. The algorithm is shown to have the same properties as its LL(1) and LR(1) counterparts.

ACKNOWLEDGEMENTS

I wish to thank Professor Charles Fischer for his constant encouragement and advice throughout my entire research work. I would also like to thank Professors Marvin Solomon and Robert Cook for their dedicated reading of the manuscript and Professors Raphael Finkel and James Smith for serving on the committee.

My fellow graduate students have made a variety of contributions to this work. I am indebted to Donn Milton, William Cox and Stephen Skedzielewski.

I am grateful to the French Government, the Computer Sciences Department at the University of Wisconsin and the National Science Foundation for their financial support.

I want to show my deepest appreciation to my friend Jane Henderson. Finally, I wish to express my gratitude to my wife Annick, my son Sébastien and my parents.

to Sébastien

TABLE OF CONTENTS

TABLE OF FIGURES

Chapter 1 : INTRODUCTION

1.1 Programming Errors

A substantial portion of a programmer's time is spent in correcting errors. These errors can be divided into several categories. Syntax error: the program cannot be generated by the grammar rules that define the programming language. We assume these rules to be defined by a grammar that includes both the usual context-free specification (BNF) and context-sensitive restrictions, sometimes called static semantics. Semantic error: the program is grammatically correct but does not conform to certain restrictions which, in general, can only be enforced at run-time (e.g., subscript out of range). Logical error: the program has a run-time effect that is different from the programmer's intention.

Automatic detection and correction of program errors can decrease the cost and enhance the quality of programs. This thesis is exclusively concerned with the automatic detection and correction of syntactic errors. However, extensive research is currently under way in all three domains. Detection of semantic errors has been studied for a long time. Some problems are now well understood (e.g. subscript checking); others were recently discovered and require more elaborate solutions (e.g. tag checking in PASCAL [FL 77]). Detection of logical errors can be achieved by using formal program verification techniques. However, it is not now possible to rigorously prove the correctness of programs of substantial size and such proof may never be common practice (see for example [DLP 77]).

Detection and correction of errors is not the only way to deal with the complex task of program development. It is also desirable to avoid having errors in the first place. Structured programming techniques [DDH 72] are intended to reduce the complexity of programming by restricting the process of creating programs. Further, good language design can provide mechanisms to minimize the presence of errors of all three kinds.

1.2 Syntactic Error Correction

The problem of correcting context-free syntax errors has received much attention. Let us distinguish between error recovery and error correction. By _error recovery_, we mean the process of restarting the parser in a valid configuration after a syntax error has been discovered. By _error correction_, we mean the process of transforming a syntactically incorrect program into a correct one.

Automatic correction of syntax errors is controversial. The argument against it is that the programmer ought to correct all errors, being the only one who knows what is really wanted. However, we think error correction can be very useful for several reasons. As noted by Holt and Barnard [HB 76], the task of _learning_ a programming language is considerably easier when the compiler produces good diagnostics. Extensive experimentation has been done with the PL/C compiler developed at Cornell University [CW 73] and the SP/k compiler developed at the University of Toronto [HB 76]. These two student-oriented compilers have been very successful in correcting syntax errors. They also try to generate code, even in the presence of minor errors, so that execution can be started, giving the student a chance to eliminate logical errors in initial runs.

Error correction can be useful in a production environment too. Although no compiler can always guess the programmer's intention, it should not skip large portions of a source program to recover from errors. Error recovery, at least, is needed to allow detection of most syntax errors in a single compilation. As we will see later, the error correction schemes we present in this dissertation can be used silently (i.e., without generation of error correction messages) to provide high quality recovery. Moreover, it appears that many common syntax errors can be readily and "correctly" repaired.

Since Irons [Iro 63] developed one of the first grammar-based error correctors, a considerable amount of work has been devoted to this domain. However, no entirely satisfactory solution has yet been found. The place of error correction within a compiler development project is discussed by Aho and Ullman [AU 77; Chapter 11]. The treatment of lexical errors falls outside the scope of this thesis. A survey of formal methods for correcting regular languages can be found in [BAC 77]. Algorithms for correcting spelling errors have been presented by Morgan [Mor 70]. We will assume that the input string has been preprocessed by a scanner, providing a token stream to the parser.

The oldest recovery scheme is called <u>panic mode</u>

recovery. When an error is detected, the parser skips input symbols until a "safe" symbol such as ";" or "end" is found. The parse stack is then erased until the safe symbol can follow the top of the parse stack. A more elaborate version of this technique is the phrase level recovery described by Leinius [Lei 70] and James [Jam 72]. The construct (i.e. phrase) currently being recognized is simply assumed to be completed, and input symbols are skipped until a symbol that can follow this construct is reached. Although they are simple and efficient, these techniques suffer very serious drawbacks. Since portions of the input program are skipped during error recovery, many compilations may be needed to remove all syntax errors. Further, it is very often the case that ill-chosen recovery induces a cascade of errors, where in fact only one error was present.

Most of the early error correction methods were essentially ad hoc. For example, error entries in a parse table were often replaced by error actions, usually "insert a terminal string" or "delete the next input symbol". This scheme was introduced by Conway and Maxwell for the CORC compiler [CM 63] and later adapted to the Cornell PL/C compiler [CW 73]. Again, such techniques have major disadvantages: their implementation has to be done by hand, they can fail on unanticipated errors, and they do not survive grammar modifications.

Aho and Peterson introduced the first error correction scheme based on a minimization model [AP 72]. Possible corrections are insertions, deletions and replacements of terminal symbols. A cost is associated with each possible correction. The total correction cost is the sum of the costs of correcting individual errors. Aho and Peterson show how this total cost can be minimized. The underlying parsing method is Earley's algorithm; errors are handled by the automatic addition of error productions to the original context-free grammar. (E.g. $a \longrightarrow \epsilon$ models a deletion error.)

Whenever an error production is used, the cost of the corresponding correction is recorded. This algorithm has obvious disadvantages. Since Earley's algorithm is used for parsing, its worst case running time is cubic in the size of the input string. Similar ideas cannot be adapted to practical parsing algorithms such as LL(1) or LR(1), because the addition of error productions renders the grammar large and ambiguous [AP 72]. However, this work has had a very important theoretical impact. Because of the fact that a minimization model is used, least-cost corrections can be obtained, not merely plausible corrections. Also, any source program can be corrected and parsed, eliminating the need for panic mode recovery that is used when the error corrector fails (e.g. [CW 73]).

The minimization model used by Aho and Peterson can be termed global, in the sense that the effect of a given correction is weighted against the whole program. Considering the fact that such an approach seems to be inherently complex, a local minimization model has been proposed by Fischer, Milton and Quiring [FMQ 77]. Since our thesis is an extension of this work to different parsing algorithms, we will present it in detail in a later section.

1.3 Definitions and Notations

In this section we review some basic definitions related to formal grammars, formal languages and parsers. We also introduce some concepts that will be needed when we discuss the correction of context-free languages.

An alphabet or vocabulary is a finite set of symbols. A sentence over an alphabet is any string of finite length composed of symbols of the alphabet. The empty sentence, ϵ, is the sentence consisting of no symbols. If Γ is an alphabet, Γ^* denotes the set of all sentences composed of symbols of Γ, including the empty sentence. Γ^{k*} denotes the set of all sentences over Γ having at most k symbols. If $x \in \Gamma^*$, $|x|$ denotes the length of x.

Context-free Grammars

Definition 1.3.1 : A context-free grammar (cfg) is a quadruple $G = (V_n, V_t, P, S)$ where

V_n is a finite set of nonterminal symbols.

V_t is a finite set of terminal symbols, disjoint from V_n.

P is a finite subset of $V_n x (V_n \cup V_t)^*$, whose elements will be denoted $A \longrightarrow X_1 \ldots X_m$. A is called the left-hand side (LHS) of the production and $X_1 \ldots X_m$ is called the right-hand side (RHS). For convenience, we assign an arbitrary but fixed numbering to the productions so that we may refer to P_i, LHS_i, RHS_i, for $i = 1, \ldots, |P|$.

S is a distinguished element of V_n, the start symbol; it does not appear on the right-hand side of any production in P. ⊠

We call $V = V_n \cup V_t$ the vocabulary of G. Any production may have an empty right-hand side. Such productions are written $A \longrightarrow \epsilon$. If $\alpha, \beta, \gamma \in V^*$ and $A \longrightarrow \beta \in P$, we say that $\alpha A \gamma$ directly derives $\alpha\beta\gamma$, denoted by $\alpha A\gamma \Longrightarrow \alpha\beta\gamma$. The reflexive and transitive closure of $\Longrightarrow$ is denoted by $\overset{*}{\Longrightarrow}$, and its transitive closure by $\overset{+}{\Longrightarrow}$.

We designate by SF(G) the set of sentential forms derivable from S, that is

$$SF(G) = \{ \alpha \in V^* \mid S \overset{*}{\Longrightarrow} \alpha \}$$

The language generated by G is $L(G) = SF(G) \sqcap V_t^*$.

A derivation $S \Longrightarrow \alpha_1 \Longrightarrow \alpha_2 \Longrightarrow \ldots \Longrightarrow \alpha_n$ is said to be a rightmost derivation if at each step the production being used is applied to the rightmost nonterminal in the sentential form α_i. We use the notations $\underset{r}{\Longrightarrow}$ and $\underset{r}{\overset{*}{\Longrightarrow}}$ for rightmost derivations. Leftmost derivations are defined in a similar manner and will be denoted by $\underset{l}{\Longrightarrow}$ and $\underset{l}{\overset{*}{\Longrightarrow}}$.

A cfg is said to be unambiguous if $w \in L(G)$ implies w has a unique rightmost (or leftmost) derivation. We say that β is a phrase of a sentential form $\alpha\beta\gamma$ if there exists a derivation $S \overset{*}{\Longrightarrow} \alpha A\gamma \overset{+}{\Longrightarrow} \alpha\beta\gamma$. β is a simple phrase if $\alpha A\gamma \Longrightarrow \alpha\beta\gamma$. A leftmost simple phrase of a sentential form is called a handle.

To every sentential form there corresponds a derivation tree. The root of the tree is S. If $A \in V_n$ is rewritten using $A \longrightarrow X_1 \ldots X_m$, then A has $X_1, \ldots, X_m$ as direct descendants. If G is unambiguous, then every sentence in L(G) has a unique derivation tree.

A cfg is said to be reduced if $\forall A \in V_n$ (1) $S \overset{*}{\Longrightarrow} \ldots A \ldots$ and (2) $\exists\, w \in V_t^*$ such that $A \overset{+}{\Longrightarrow} w$.

Given a cfg $G = (V_n, V_t, P, S)$, the corresponding augmented grammar is $G' = (V_n \cup \{S'\}, V_t \cup \{\$\}, P \cup \{S' \longrightarrow \$S\$\}, S')$, where S' and \$ (the end marker) are new symbols not in V. Let us denote $V_t \cup \{\$\}$ by $\hat{V}_t$ and $V_n \cup \{S'\}$ by $\hat{V}_n$. Also let $\hat{V} = \hat{V}_t \cup \hat{V}_n$. From now on we will assume, without explicit mention, that all grammars are reduced and have been so augmented.

The set of terminal symbols that may follow $U \in V$ in some sentential form of G is given by

$$\text{Follow}^G(U) = \{\, a \in \hat{V}_t \mid S' \overset{*}{\Longrightarrow} \ldots Ua \ldots \,\}$$

The set of terminal symbols that are a one-symbol (or less) prefix of a terminal string derivable from $\alpha \in \hat{V}^*$ is given by

$$\text{First}^G(\alpha) = \{\, a \in \hat{V}_t^{1*} \mid (\alpha \overset{*}{\Longrightarrow} aw,\ w \in \hat{V}_t^*,\ a \in \hat{V}_t) \ \underline{\text{or}}\ (\alpha \overset{*}{\Longrightarrow} a \ \underline{\text{and}}\ a = \epsilon) \,\}$$

In the case G is clearly understood, we use Follow(U) and First(α).

Unless otherwise specified, the following conventions for naming strings will be used throughout this dissertation:

a, b, c,... are members of $\hat{V}_t$.

A, B, C,... are members of $\hat{V}_n$.

U, V, W,... are members of $\hat{V}$.

α, β, γ,... are members of $\hat{V}^*$.

u, v, w,... are members of $\hat{V}_t^*$.

Context-free grammars have been studied in detail. Properties of context-free grammars can be found in [HU 69] and [AU 73]. A notation similar to that of PASCAL [JW 75] is rather extensively used to describe algorithms.

Context-free Error-Correcting Parsers

Definition 1.3.2 : A parser is an algorithm that given a cfg G and $w \in \hat{V}_t^*$

(1) signals an error if $w \notin L(G)$

(2) otherwise determines a derivation tree of w. ⊠

A left parse of w is the sequence of productions used in a lefmost derivation of w. A right parse of w is the reverse of the sequence of productions used in a rightmost derivation of w. Either a left or a right parse can be used to uniquely specify a derivation tree.

In this dissertation, we restrict our attention to parsers that can deterministically produce a left parse or a right parse by a single scan of the input, from left to right. The LL grammars are those that can be parsed "naturally" (i.e., in a single left-to-right parse using fixed lookahead) by a deterministic left parser. The LR grammars are those that can be parsed "naturally" by a deterministic right parser. The theory of LL and LR parsers can be found in [AU 73], with some additional material in [DeR 71].

We now consider <u>error-correcting parsers</u> for a cfg G that can be based on these classes of parsers. A correction will be performed upon detection of a syntax error. Assume $w = \$xa_1 \ldots a_n\$$ is such that $S' \overset{+}{\Longrightarrow} \$xy$ for some $y \in \hat{V}_t^*$, but there is no $z \in \hat{V}_t^*$ such that $S' \overset{+}{\Longrightarrow} \$xa_1z$, and an error is detected by the left-to-right parser upon first seeing a_1, termed the <u>error symbol</u>. The error corrector restarts the parser by <u>deleting</u> $a_1 \ldots a_i$, $\emptyset \leq i \leq n$ and <u>inserting</u> $y \in V_t^*$ such that $S' \overset{+}{\Longrightarrow} \$xya_{i+1} \ldots$ † . Such error correcting parsers are capable of correcting and parsing <u>any</u> input

---

† The notation $S \overset{+}{\Longrightarrow} x \ldots$ or $x \ldots \in L(G)$ means there is some $y \in \hat{V}_t^*$ such that $xy \in L(G)$.

string if and only if the parser has the correct prefix property, that is if the sequence of symbols to the left of the erroneous symbol is always the prefix of some $w' \in L(G)$ [LRS 76]. With such schemes, symbols that have been accepted by the parser can be considered unchangeable, since the error corrector does not tamper with the left-context. Therefore, a one-pass compiler never has to "back up" for semantic and code generation purposes.

In order for the error corrector to succeed, we require a parser that will never make a (possibly incorrect) move when an erroneous input symbol appears as the lookahead. Such a parser is said to have the immediate error detection property (IEDP) [FTM 78]. It is well-known that full LL(1) and LR(1) parsers have the IEDP. However, practical variations such as Strong LL(1), SLR(1), LALR(1) do not. We will investigate modifications to these algorithms so that the IEDP holds.

We assume a given insertion cost function IC supplied with $IC(\epsilon) = 0$ and $IC(a) \geq 0$ for all $a \in \hat{V}_t$. We introduce a special symbol "?" such that $IC(?) = \infty$. IC($) is assigned an arbitrary but very large value (say, 10,000) because $ can never be inserted during error correction (it is always correctly provided as an end marker). We intentionally avoid setting IC($) to infinity to ensure that the concept of a least-cost string is always well-defined (e.g. $IC(a\$) > IC(\$)$). Also, let the deletion cost function $DC(a) \geq \emptyset$ for $a \in V_t$ † be the cost of deleting a. We define $IC(a_1 \ldots a_n)$ to be $IC(a_1) + \ldots + IC(a_n)$ and $DC(a_1 \ldots a_n)$ to be $DC(a_1) + \ldots + DC(a_n)$. We now formally define locally least-cost error correction as a minimization problem.

<u>Definition 1.3.3</u> : Given an input string $\$xa_1 \ldots a_n\$$ such that $\$x \ldots \in L(G)$ but $\$xa_1 \ldots \notin L(G)$, a <u>locally least-cost error corrector</u> finds an optimal solution (i,y) to

$$\min_{\emptyset \leq i' \leq n} \{ \min_{y' \in V_t^*} \{DC(a_1 \ldots a_{i'}) + IC(y') \mid \$xy'a_{i'+1} \ldots \in L(G)\}\}$$ ☒

Throughout this dissertation, we will only consider <u>locally least-cost</u> correctors. The class of corrections we include in our minimization model is kept intentionally simple (replacements and transpositions are not considered). More complete models seem to unduly complicate the correction process and do not appear necessary to obtain useful error correctors.

---

† DC($) need not be defined since "$" will never be involved in a deletion.

At times we will even consider a simpler model where a correction is always done solely by insertion of a terminal string.

<u>Definition 1.3.4</u> : Given an input string \$xa... such that \$x... $\in$ L(G) but \$xa... $\notin$ L(G), an <u>insertion-only locally least-cost error corrector</u> finds an optimal solution to

$$\min_{y' \in V_t^+} \{ IC(y') \mid \$xy'a... \in L(G) \}$$ ☒

This later model applies to a restricted class of grammars termed <u>insert-correctable</u> cfg's for which it is guaranteed that the optimal solution to the above problem is never "?" [FMQ 77].

<u>Definition 1.3.5</u> : A cfg G is said to be <u>insert-correctable</u> if and only if $\forall$ x $\in$ V_t^* and a $\in$ $\hat{V}_t$ such that \$x... $\in$ L(G) but \$xa... $\notin$ L(G) there exists y $\in$ V_t^+ such that \$xya... $\in$ L(G). ☒

Even if G is insert-correctable, the method of Definition 1.3.3 may yield a lower cost correction than that of Definition 1.3.4, but it will require a more complicated corrector.

1.4 An Error Corrector for LL(1) Parsers

LL(1) Parsing

LL(1) grammars are those for which a parser generating a left parse can operate deterministically, assuming it is allowed to look at one input symbol to the right of its current input position. Informally, an LL(1) parser works in the following fashion: given a left-sentential form $wA\alpha$ that occurs during the parse of wx (where w has already been processed), the parser can always decide (by construction) which production was used to expand A knowing only the first symbol of x. The LL(1) parsing algorithm uses a push-down stack and a parsing table †

$$M : \hat{V}_n \times \hat{V}_t \longrightarrow \{ \underline{predict}\ i \mid 1 \leq i \leq |P'| \} \cup \{ \underline{error} \}$$

where M(A, a) = predict i says production P_i has to be used to expand A when "a" is the next input symbol. The parse stack contains the part of the current left-sentential form that has not yet been expanded (in the above case $A\alpha$). The theory of LL(1) grammars can be found in [AU 73].

---

† Many models assume M is defined over $\hat{V} \times \hat{V}_t$ and include pop and accept in the range of M. However, this expanded table is not necessary in practice.

LL(1) Error Correction

A locally least-cost error corrector using only insertions for LL(1) parsers was presented by Fischer, Milton and Quiring in [FMQ 77]. We will refer to it as the FMQ algorithm.

The error corrector requires the immediate error detection (IEDP) property. Commonly used Strong LL(1) parsers don't have the IEDP since productions may be erroneously predicted when an error symbol is seen as the lookahead. The FMQ algorithm uses a Full LL(1) parser, which guarantees the IEDP. (The only difference between Strong and Full LL(1) parsers is in the size and complexity of the parsing tables.) However the IEDP can be obtained in Strong LL(1) parsers as is discussed in [FTM 78].

As noted above the original FMQ algorithm operates by insertion only. Just as parsers are driven by precalculated parsing tables, the error corrector is driven by error correction tables that can be computed in advance given a cfg G and IC, the insertion-cost function. Least-cost terminal strings that can be derived from vocabulary symbols are tabulated in S. Least-cost terminal prefixes that allow an error symbol to be generated from vocabulary symbols are tabulated in E.

(1) For $X \in \hat{V}$, we define $S(X)$ to be an optimal solution to

$$\min_{y \in \hat{V}_t^*} \{ IC(y) \mid X \overset{*}{\Longrightarrow} y \}$$

Also, we define $IC(\alpha) = IC(S(\alpha))$ and $S(X_1 \ldots X_n) = S(X_1) \ldots S(X_n)$.

(2) For $A \in \hat{V}$ and $a \in \hat{V}_t$, we define $E(A, a)$ to be a solution to

$$\min_{y \in V_t^*} \{ IC(y) \mid A \overset{*}{\Longrightarrow} ya\ldots \}$$

If $A \overset{*}{\not\Longrightarrow} \ldots a \ldots$, we set $E(A, a) = ?$.

Both S and E tables (and variations of them) will be used for all the error correctors we develop in this dissertation. Algorithms for computing these tables are given in Appendix A.1 .

The error corrector (see Figure 1.4.1) operates very simply by considering the symbols on the parse stack. It computes a least-cost string to be inserted to the immediate left of the error symbol "a" to allow "a" to be accepted by the parser.

```
function LL_Insert(σ, a): TerminalString;
   σ = X1...Xn, the parse stack;
   a ∈ V̂t, the error symbol;
begin
  1  Insert := ∈;
  2  for i from 1 to n do
  3     if E(Xi,a) = ?
  4        then Insert := Insert cat S(Xi)
  5        else return ( Insert cat E(Xi,a) )
  6     fi
  7  od;
  8  return( ? )  (* failure *)
end LL_Insert.
```

<u>Figure 1.4.1</u> : function LL_Insert

The return(?) statement in line 8 is needed only in the case G is not insert-correctable. The procedure scans down the stack until it finds the <u>first</u> stack symbol that can generate the error symbol. LL_Insert does not necessarily return the least-cost insertion (in the sense of Definition 1.3.4) since, in some cases, a lower cost correction can be generated by considering symbols further down the stack. Properties of LL_Insert are investigated in [FMQ 77]. In particular, it is shown that parsing and correcting any string in V_t^* can be done in time linear with respect to the length of this string. An extended FMQ algorithm is presented in [FM 77]. It includes extended stack search and deletions. This modified algorithm is a locally least-cost error corrector as in Definition 1.3.3 .

1.5 Organization of the Thesis

This dissertation describes extensions of the FMQ error corrector to various parsing algorithms. Chapter 2 contains a brief summary of the theory of LR(1) parsing and then develops a locally least-cost error corrector for LR(1) parsers that operates by insertion only. Chapter 3 extends the error corrector to include deletions as well. The properties of the algorithm are presented. In particular, correctness is established and linearity is shown in cases of special interest. Implementation results are discussed. Chapter 4 introduces the notion of context-sensitive error correction. A formulation of Attributed LL(1) grammars (AF-LL(1)) that was developed by Watt [Wat 77a] is presented and a locally least-cost error corrector for a restricted class of AF-LL(1) grammars is developed. Chapter 5 discusses the significance of this work and gives directions for future research.

Chapter 2 : AN INSERTION-ONLY ERROR CORRECTOR FOR LR(1)-BASED PARSERS

In this chapter, we briefly review the LR(1) parsing techniques and we then develop a locally least-cost error corrector for LR(1)-based parsers that operates only by insertion. The class of LR(1)-based parsers includes the LR(0), SLR(1), LALR(1) and canonical LR(1) parsers. We assume that all cfg's considered in this chapter are insert-correctable, so that the model of Definition 1.3.4 is applicable.

2.1 LR(1) Parsing

LR(1) grammars are those for which a parser generating a right parse can operate deterministically assuming it is allowed to look at most one input symbol to the right of its current input position. More formally, an augmented cfg G is LR(1) if the conditions

(1) $S' \overset{*}{\underset{r}{\Longrightarrow}} \alpha A w \underset{r}{\Longrightarrow} \alpha\beta w$

(2) $S' \overset{*}{\underset{r}{\Longrightarrow}} \gamma B x \underset{r}{\Longrightarrow} \gamma\delta x = \alpha\beta y$

and (3) First(w) = First(y)

imply that $\alpha = \gamma$, A = B and x = y.

Definition 2.1.1 : An LR(1)-based parser for an augmented cfg $G = (\hat{V}_n, \hat{V}_t, S', P')$ is a four-tuple ($\mathbf{S}$, $s_\emptyset$, GOTO, PA) where

$\mathbf{S}$ is a finite set of states.

$s_\emptyset$ is a distinguished element of $\mathbf{S}$, the start state.

GOTO is the transition function, a mapping from $\mathbf{S} \times \hat{V}$ into $\mathbf{S} \cup$ { error }.

PA is the parsing action function, a mapping from $\mathbf{S} \times \hat{V}_t$ to { shift, accept, error } $\cup$ { reduce i | $1 \leq i \leq |P'|$ }. ⊠

The operation of an LR(1)-based parser can be characterized by its action on parsing configurations. A configuration is a pair (σ, w) where $\sigma = s_\emptyset s_1 \ldots s_p$, $s_i \in \mathbf{S}$, $i = \emptyset, \ldots, p$ is a stack of states called the parse stack (s_p is the top-most state) and $w \in \hat{V}_t^*$ is the remaining input string. The initial configuration is ($s_\emptyset$, x) where x is the input string. The parser moves from one configuration to the next by shifting or reducing. The transition relation is denoted by $\vdash$. Given some configuration $(\sigma s_p, aw)$ we have :

(1) $(\sigma s_p, aw) \vdash (\sigma s_p s_{p+1}, w)$ if $PA(s_p, a) =$ <u>shift</u> and $GOTO(s_p, a) = s_{p+1}$.

(2) $(\sigma s_p, aw) \vdash (s_\emptyset s_1 \ldots s_q, aw)$ if $PA(s_p, a) =$ <u>reduce</u> i. $s_\emptyset s_1 \ldots s_q$ is obtained from σs_p by removing $|RHS_i|$ symbols from σs_p, giving $s_\emptyset \ldots s_{q-1}$, and pushing $s_q = GOTO(s_{q-1}, LHS_i)$ onto the stack. Production number i is output as part of the parse of the input string.

(3) $(\sigma s_p, aw)$ is an <u>error</u> <u>configuration</u> if $PA(s_p, a) =$ <u>error</u>. In this case error recovery has to take place before the parser can be restarted. Recovery is done by transforming σs_p and aw so that a valid configuration (i.e. a configuration which does not yield <u>error</u>) is obtained.

(4) $(\sigma s_p, \$)$ is an accepting configuration if $PA(s_p, \$) =$ <u>accept</u>. In this case the parser halts and the output generated is the right parse of the (possibly corrected) input string.

We define $GOTO(s, U_1 \ldots U_n) = GOTO(GOTO(s, U_1), U_2 \ldots U_n)$ for $s \in \mathbb{S}$, $U_1 \ldots U_n \in \hat{V}^*$ and $n > 1$. We now consider the construction of an LR(1)-based parser for a given augmented cfg G. Following DeRemer [DeR 71], we first consider cases where a parser can be generated by constructing the LR(Ø)-machine or characteristic finite state machine (CFSM) of G. The CFSM is a deterministic finite automaton which recognizes the viable prefixes of G. $\gamma \in \hat{V}^*$ is a viable prefix of G' if there exists a derivation $S' \overset{*}{\underset{r}{\Longrightarrow}} \alpha A w \underset{r}{\Longrightarrow} \alpha\beta w$ such that γ is a prefix of $\alpha\beta$ (that is a prefix of a right sentential form which does not extend past the handle).

We call $\mathbb{S}$ the set of states of the CFSM and GOTO: $\mathbb{S} \times \hat{V} \longrightarrow \mathbb{S}$ its transition function. A state is a set of LR(Ø)-items. An LR(Ø)-item for G is an object of the form $[A \longrightarrow \alpha \bullet \beta]$ where $A \longrightarrow \alpha\beta \in P$; β is called the trailing part of the item. A production $A \longrightarrow \epsilon$ generates only one item $[A \longrightarrow \bullet]$. An item of the form $[A \longrightarrow \alpha\bullet]$ is called a final item. Item $[A \longrightarrow \beta_1 \bullet \beta_2]$ is valid for $\alpha\beta_1$, a viable prefix of G, if there exists a derivation $S' \overset{*}{\underset{r}{\Longrightarrow}} \alpha A w \underset{r}{\Longrightarrow} \alpha\beta_1\beta_2 w$. For a right sentential form $\alpha\beta$, $\alpha\bullet\beta$ denotes this sentential form with α selected as a viable prefix (of $\alpha\beta$). The start state $s_\emptyset$ contains an item $[S' \longrightarrow \$\bullet S\$]$. (We assume that the left-end marker is consumed in advance.)

Considering a state $s \in \mathbf{S}$, we partition s into its basis and closure sets:

$$basis(s) = \{ I \in s \mid I = [A \longrightarrow \alpha \bullet \beta] \text{ and } \alpha \neq \epsilon \}$$

$$closure(s) = \{ I \in s \mid I = [B \longrightarrow \bullet \gamma] \}$$

We will see that the parser will enter states corresponding to item sets of the CFSM. The state indicates which part of which productions may have been recognized. (The part which may be recognized is to the left of the $\bullet$.) An algorithm that computes the CFSM is presented in Appendix A.2 .

The CFSM is then used to construct an LR(1)-based parser, as will be outlined below. In fact, we only have to compute PA, the parsing action function. All LR(1)-based parsing techniques are alike in that they all use the PA and GOTO functions in exactly the same way. Different schemes have been devised for sub-classes of the LR(1) grammars. Let us briefly discuss the LR(Ø), SLR(1), LALR(1) and full LR(1) grammars.

A state $s \in \mathbf{S}$ is <u>LR(Ø)-inadequate</u> if it contains two items of the form $[A \longrightarrow \alpha \bullet]$ and $[B \longrightarrow \beta \bullet]$ (a <u>reduce-reduce</u> conflict) or two items of the form $[A \longrightarrow \alpha \bullet a \delta]$ and $[B \longrightarrow \beta \bullet]$ (a <u>shift-reduce</u> conflict). If $\mathbf{S}$ does not contain any inadequate state, then G is LR(Ø) and PA is trivially obtained from the item sets (see, for example, [DeR 71]). However it is very difficult to write an LR(Ø) grammar for a realistic programming language.

When an LR(Ø)-inadequate state is entered, we don't know which action to take. The Simple LR(1) technique gives a solution to this conflict [DeR 71]. An augmented cfg G is SLR(1) if and only if for all LR(Ø)-inadequate states we have

(1) $[A \longrightarrow \alpha \bullet X\beta] \in s$ and $[B \longrightarrow \gamma \bullet] \in s$ implies

Follow(B) $\sqcap$ EFirst(Xβ cat Follow(A)) = $\emptyset$, where EFirst(δ) = { $a \in V_t$ | $\delta \overset{*}{\underset{r}{\Longrightarrow}} ax$ and $Dw \underset{r}{\Longrightarrow} w$ is never a step in the derivation of ax}. In this case we are able to resolve a shift-reduce conflict.

(2) $[A \longrightarrow \alpha \bullet] \in s$ and $[B \longrightarrow \gamma \bullet] \in s$ implies

Follow(A) $\sqcap$ Follow(B) = $\emptyset$. In this case we are able to resolve a reduce-reduce conflict.

The SLR(1) grammars are powerful enough to specify the syntax of most common programming languages, such as PASCAL [JW 75]. However, more flexibility is given at little extra cost by the class of Lookahead LR(1) (or LALR(1)) grammars.

An LALR(1) parser constructor uses <u>local lookahead</u> information instead of First and Follow sets to determine

PA. Starting with the CFSM as previously defined, a set of valid lookahead symbols is attached to every item in every state † . LALR(1) grammars are powerful enough to allow for an easy specification of most, if not all, the context-free aspects of common programming languages constructs.

On the other hand, a canonical LR(1) parser can be generated by constructing the LR(1)-machine where the lookahead is built into the items [AU 73; Volume 1]. (An LR(1)-item is an object of the form $[A \longrightarrow \alpha \bullet \beta, a]$ where $a \in \hat{V}_t^{1*}$.) Canonical LR(1) parsers are of theoretical interest only, because of the size of the machine. An LALR(1) parser for a language such as PASCAL may have 200 states; the corresponding LR(1) parser may have several thousand states [AU 77]. A number of schemes for reducing the size of LR(1) parsing tables have been developed [AU 73; Volume 2]. However these are irrelevant in practice because LR(1)'s extra power is not needed for common programming languages. Since the lookahead component of an LR(1)-item is irrelevant to the error correctors we will develop, it will simply be ignored in all of the following discussions.

In summary, we have isolated a class of LR(1)-based

---

† See, for example, [AU 77] for a detailed construction.

parsers that only differ in the way the parsing action function is obtained from the LR(Ø) or LR(1)-machine. The error correctors we develop in Chapters 2 and 3 are defined in the same way for all LR(1)-based parsers.

2.2 Immediate Error Detection

It is a well-known fact that canonical LR(1) parsers have the immediate error detection property [AU 73]. However none of the other LR(1)-based parsers we discussed have the IEDP. Because of the use of approximations to exact lookaheads, it is possible to do some incorrect reductions when using an erroneous symbol as the lookahead.

<u>Example 2.2.1</u> : Consider the following grammar G_1 †

1. $S' \longrightarrow \$E\$$	2. $E \longrightarrow TE'$
3. $E' \longrightarrow *TE'$	4. $E' \longrightarrow \epsilon$
5. $T \longrightarrow b$	6. $T \longrightarrow [\ E\ ]$

---

† G_1 generates all infix expressions using * as operator, "b" as operand and [] as parentheses (e.g. $[b*b]*b$).

Part of G_1's CFSM is as follows:

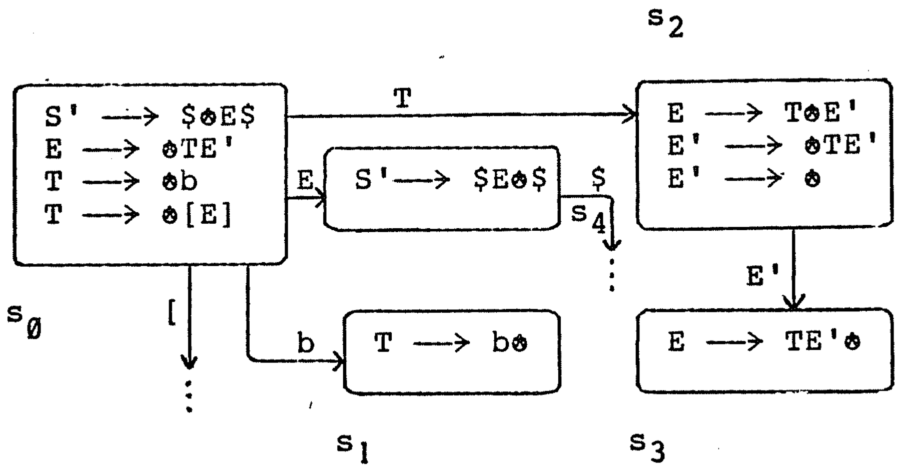


Figure 2.2.1 : G_1's CFSM

Now assume an SLR(1) parser constructor has been used and try to parse "$b]$". The parse is:

($s_\emptyset$, "b]$")	$\vdash$ ($s_\emptyset s_1$, "]$")	output ($T \longrightarrow b$)
	$\vdash$ ($s_\emptyset s_2$, "]$")	output ($E' \longrightarrow \epsilon$)
	$\vdash$ ($s_\emptyset s_2 s_3$, "]$")	output ($E \longrightarrow TE'$)
	$\vdash$ ($s_\emptyset s_4$, "]$")	error

It is now too late for an insertion-only error corrector to do a correction; in fact at this point no insertion at all is possible. The parser made an erroneous reduction $E' \longrightarrow \epsilon$ because "]" $\in$ Follow(E'). ⊠

Assume (σs_p, aw) is an error configuration (that is,

$PA(s_p, a)$ = error). We need to restore the parser configuration to what it was at the time "a" first appeared as the lookahead. It is clear that the only parser moves an error symbol can induce are reductions. Therefore stack restoration can be obtained in the following way:

> Whenever a new symbol $a \in \hat{V}_t$ is used as the lookahead, initialize an auxiliary stack AS to an empty stack. From now on record each reduction in the auxiliary stack. If a syntax error is detected while "a" is the lookahead, restore the parse stack to the state it had when "a" became the lookahead symbol, using information kept in AS. Given $\sigma = s_1 \ldots s_i$ and reduce j where $P_j = (A \longrightarrow U_1 \ldots U_m)$, the reduction is undone by making $\sigma = s_1 \ldots s_{i-1} s'_1 \ldots s'_m$ where $s'_1 = GOTO(s_{i-1}, U_1)$ and $s'_p = GOTO(s'_{p-1}, U_p)$ for $p = 2, \ldots, m$. (This operation will subsequently be denoted by "restore(σ, AS)".) If "a" is accepted by the parser (i.e., scanned), clear the auxiliary stack AS for the next lookahead symbol.

In the above example the parser configuration would be restored to ($s_\emptyset s_2$, "]\$"). After stack restoration, the recovered parse stack can be used to drive the error correction process, as will be detailed in the following sections. We now discuss the efficiency of stack restoration.

Theorem 2.2.1 : Assume an LR(1)-based locally least-cost error correcting parser processes $\$x\$$. Then stack restoration requires at most $O(|x|^2)$ time and $O(|x|)$ space.

Proof : At any time, the size of AS is bounded by a constant times the size of the parse stack, whose maximum height is $O(|x|)$. Therefore, a given restoration can be done in time $O(|x|)$. Moreover the number of restorations will be bounded by $|x|$. This is because every input symbol is potentially erroneous and inserted symbols never cause any invocation of restore (the inserted string is correct and allows the error symbol to be accepted). The $O(|x|)$ space bound follows directly from the $O(|x|)$ maximum height of AS. ☒

In the worst case, stack restoration alone can render the error correcting parser non-linear. However, in the case of typical programming languages, there are strong reasons to believe this algorithm will require at most a number of steps bounded by a rather small constant: only right recursion in a production (direct or indirect) can make stack restoration at times require more than a constant number of moves. However, right recursion is almost invariably avoided in LR parsers (in favor of left recursion) precisely because it increases the parse stack depth required to parse various constructs.

Moreover, in a later section we will prove linearity of stack restoration in the case of a bounded depth parse stack. This property is very desirable since a linear-time error correcting parser for bounded depth parse stack, LR(1)-based parsers having the IEDP will be developed.

2.3 Right Context of an Item

In order to be able to do a least-cost insertion to the immediate left of an error symbol we need to know which strings can appear to the right of the last accepted symbol. This same problem is easier in the LL(1) error corrector [FMQ 77] since, in that case, what we expect to see is stored explicitly in the parse stack.

In the LR(1)-based case, the right context will be used to characterize this set of strings.

Definition 2.3.1 : Consider an item $I = [A \longrightarrow \beta_1 \bullet \beta_2]$ and a parse stack $\sigma = s_\emptyset \ldots s_p$ corresponding to a viable prefix $\alpha\beta_1$ for which I is valid. The right context of (I, σ) is characterized by a set of strings $R(I, \sigma)$ in

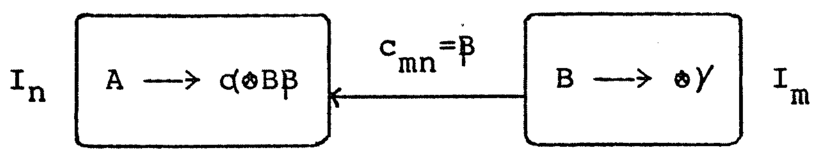
V^* such that

(1) for all $w \in V_t^*$, $S' \overset{*}{\underset{r}{\Longrightarrow}} \alpha\beta_1\beta_2 w$ implies there exists $\gamma \in R(I, \sigma)$ such that $\gamma \overset{*}{\Longrightarrow} w$.

(2) for all $\delta \in R(I, \sigma)$, we have $S' \overset{*}{\Longrightarrow} \alpha\beta_1\beta_2\delta$. ⊠

We now consider the problem of computing $R(I, \sigma)$. It will be written as a regular expression over $\hat{V}$. We first consider the <u>local right context</u>, which is that part of $R(I, \sigma)$ which can be obtained by considering only the predictions used in computing the closure set of the top stack state s_p (independently of the rest of σ).

In order to compute the local right context, we define the <u>closure graph</u> $Cl(s)$ of a state $s \in \mathbb{S}$. Nodes of $Cl(s)$ are items in s. If we obtain item $I_m = [B \longrightarrow \bullet\gamma]$ from item $I_n = [A \longrightarrow \alpha\bullet B\beta]$, we put an edge (I_m, I_n) in $Cl(s)$ and label it $c_{mn} = \beta$. β is that part of the local right context of I_m that comes from I_n by making a prediction:



<u>Figure 2.3.1</u> : closure graph construction

<u>Example 2.3.1</u> : Consider the following SLR(1) insert-correctable grammar G_2, which will be used in all the examples that follow †.

1. $S \longrightarrow \$E\$$

2. $E \longrightarrow E+T$ 3. $E \longrightarrow T$

4. $T \longrightarrow a$ 5. $T \longrightarrow (E)$

We first build part of G_2's CFSM, which will be needed in later examples.

---

† G_2 generates all infix arithmetic expressions using + as operator, "a" as operand and () as parentheses.

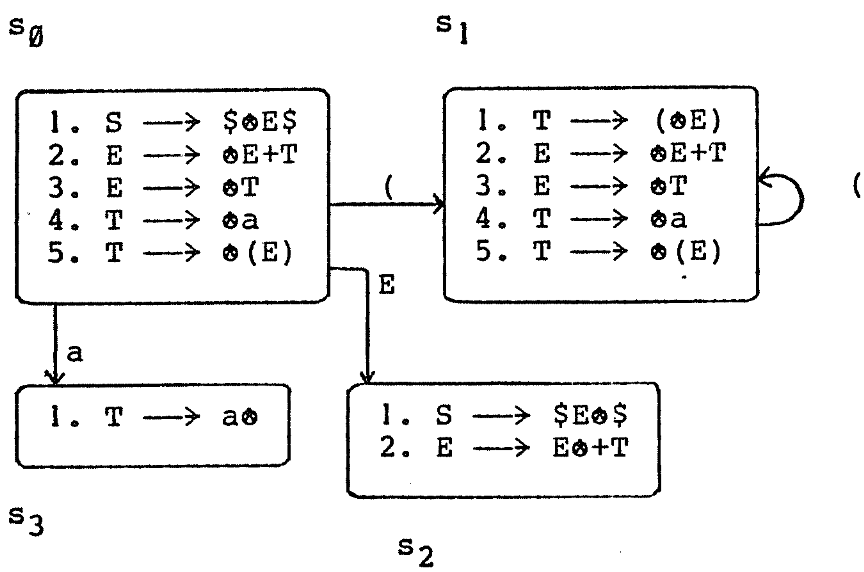


Figure 2.3.2 : G_2's CFSM

We now build the closure graph of state $s_\emptyset$.

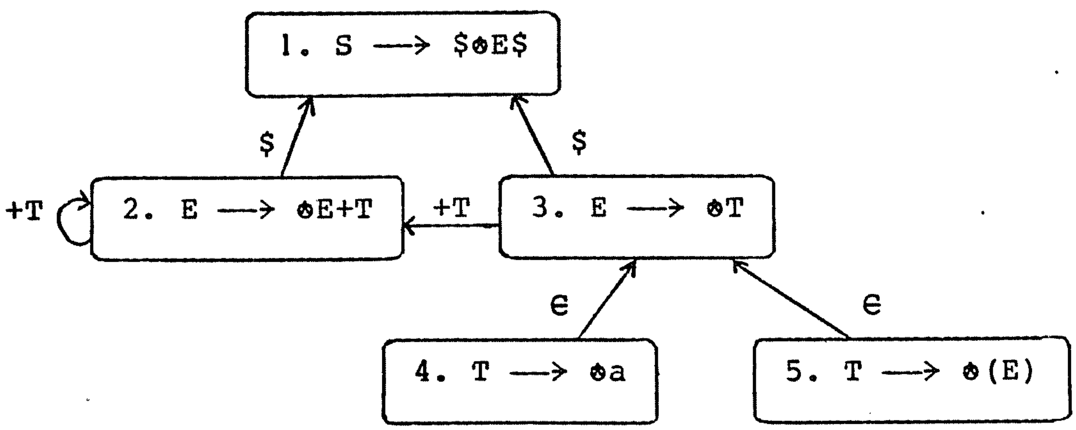


Figure 2.3.3 : closure graph $Cl(s_\emptyset)$

<u>Definition 2.3.2</u> : lc(I, s) will denote the regular set of all paths from I to any basis item in Cl(s) and lc(I, J, s) will denote the regular set of all paths from item I to item J in Cl(s). Also, L(lc(I, s)) is the set of all terminal strings derivable from members of the regular set denoted by lc(I, s). L(lc(I, J, s)) is similarly defined. ⊠

Let $\{I_k, k = 1$ <u>to</u> $|s|\}$ be the set of items in state s. In order to compute lc(I_k, s), we need to consider all paths between I_k and any item in basis(s). Each time an edge (i,j) is used on a path, c_{ij} is concatenated to the local right context. We can use the "all paths" algorithm given in [AHU 76; p.198] to compute lc(I_i, s), for all I_i's. In lines 1-11 of the following procedure LocalContext(s), we compute t_{ij}^k for all $1 \leq i \leq |s|$, $1 \leq j \leq |s|$ and $\emptyset \leq k \leq n$. t_{ij}^k is obtained by concatenation of the labels of all paths from I_i to I_j such that all nodes on the path, except possibly the end points, are in the set $\{I_1,\ldots,I_k\}$ for $k > \emptyset$. For $k = \emptyset$, we do not allow any intermediate node.

```
procedure LocalContext(s);
begin
 1  for i, j from 1 to |s| do
 2      t^Ø_ij := if ∃ (i,j) ∈ Cl(s) then {c_ij} else ∅;
 3  od;
 4  for i from 1 to |s| do
 5      t^Ø_ii := t^Ø_ii ∪ {ε}
 6  od;
 7  for k from 1 to |s| do
 8      for i, j from 1 to |s| do
 9          t^k_ij := t^(k-1)_ij ∪ (t^(k-1)_ik cat (t^(k-1)_kk )* cat t^(k-1)_kj )
10      od
11  od;
12  for i from 1 to |s| do
13      lc(I_i,s) := ∪ { t^|s|_ij | I_j ∈ basis(s) }
14  od
end LocalContext.
```

Figure 2.3.4 : procedure LocalContext

Note that, after execution of the for loop in lines 7-11, we have $lc(I_i, I_j, s) = t_{ij}^{|s|}$. In the above example we obtain:

$lc(I_1, s_\emptyset) = \{\epsilon\}$

$lc(I_k, s_\emptyset) = \{+T\}^*$ cat $\{\$\}$ for $k = 2,3,4,5$

$lc(I_3, I_2, s_\emptyset) = \{+T\}^*$

The right context $R(I, \sigma)$ can now be obtained by concatenating appropriate local right contexts for each state on the parse stack:

(1) If $I \in closure(s_p)$

$$R(I, \sigma) = \cup \{ lc(I, J, s_p) \text{ \underline{cat} } R(J, \sigma) \mid J \in basis(s_p) \}$$

(2) If $I = [A \longrightarrow \alpha X \bullet \beta] \in basis(s_p)$

$$R(I, \sigma) = \underline{\text{if}}\ p = 0 \ \underline{\text{then}}\ \epsilon \ \underline{\text{else}}\ R([A \longrightarrow \alpha \bullet X\beta], s_0 \ldots s_{p-1})$$

__Theorem 2.3.1__ : Given an item $I = [A \longrightarrow \beta_1 \bullet \beta_2]$ and a parse stack $\sigma = s_0 \ldots s_p$ corresponding to a viable prefix $\alpha\beta_1$ for which I is valid, the above algorithm computes a regular expression $R(I, \sigma)$ corresponding to Definition 2.3.2 .

__Proof__ : (Outline) By construction of the local right context, we know that $lc(I, J, s_p)$ is a regular expression. Since $R(I, \sigma)$ is obtained by a finite number of applications of steps (1) and (2) where union and concatenation operate on lc expressions, it follows that $R(I, \sigma)$ is a regular expression.

The proof that $R(I, \sigma)$ corresponds to Definition 2.3.2 is by induction on the number of times steps (1) and (2) are applied. A similar proof is given in the next section for a restricted case (see Lemma 2.5.1), so it is not detailed here. ☒

<u>Example 2.3.2</u> : Reconsider Example 2.3.1 . Now assume we want to compute the right context of item I_1 = [T ⟶ a•] in state s_3, assuming the parse stack is $s_\emptyset s_3$. We obtain:

R([T ⟶ a•], $s_\emptyset s_3$)

= R([T ⟶ •a], $s_\emptyset$)

= lc([T ⟶ •a], [S ⟶ $•E$], $s_\emptyset$)

<u>cat</u> R([S ⟶ $•E$], $s_\emptyset$)

= $\{+T\}^*$ <u>cat</u> {$} <u>cat</u> R([S ⟶ $•E$], $s_\emptyset$)

= $\{+T\}^*$ <u>cat</u> {$} ☒

In the next section, we will only consider those strings derivable from a right context that have a chance to be used in a least-cost insertion. Higher-cost insertions are of no interest for our purposes.

2.4 The Error Corrector

We are now ready to present an error corrector for LR(1)-based parsers. Its input is $a \in \hat{V}_t$, the error symbol and $\sigma = s_1 \ldots s_p$, the (restored) parse stack; its output is a least-cost insertion string that allows the error symbol to be accepted by the parser (Definition 1.3.4). Our algorithm requires the computation of an error correction table for each state $s \in \mathbf{S}$. These tables can be computed at the time the parser is generated.

Error Correction Tables

(1) Let $I_k = [A \longrightarrow \alpha \bullet \beta] \in$ basis(s)

(a) If the error symbol $a \in \hat{V}_t$ is not generable from β, we may need to insert the least-cost string derivable from β, so we need to tabulate $S(\beta)$.

(b) The least-cost insertion that will allow $a \in \hat{V}_t$ to be generated via $\beta = U_1 \ldots U_n$ $(n \geq 0)$ is $S(U_1 \ldots U_i)$ cat $E(U_{i+1}, a)$ where i minimizes $IC(S(U_1 \ldots U_i)$ cat $E(U_{i+1}, a))$. Call this string Insert(β, a). If $\beta \overset{*}{\not\Rightarrow} \ldots a \ldots$ then Insert(β, a) = ? .

(c) In the event no optimal insertion can be generated from state s we will have to generate insertions based upon state s', an immediate predecessor of s. Therefore we need to tabulate the list of possible predecessors of I_k. Elements of this list will be pairs (m, s') such that state s' is an immediate predecessor of s in the CFSM and $I_m \in s'$ is the item which produced I_k. Call this list Pred(I_k). It is trivially obtained by extending procedure CFSM (Appendix A.2).

(2) Let $I_k = [A \longrightarrow \bullet\gamma] \in$ closure(s)

(a) For such an item we need to tabulate the list of backpointers to all items in basis(s) that can be reached from it in the closure graph, Cl(s). Each backpointer is a pair (m,y) where $I_m \in$ basis(s) and y is a least-cost terminal string that can be used to reach I_m from I_k. Call this list B(I_k).

In Example 2.3.1 (Figure 2.3.3), we have B(I_5) = {(1,\$)}; basis item I_1 is the only one reachable from I_5 in Cl($s_\emptyset$), and the least-cost terminal string that can be used along a path (I_5,...,I_1) is "\$" .

B(I_k) for all k such that $I_k \in$ closure(s) can be obtained by using a shortest-path algorithm on Cl(s) where the cost of using edge (i, j) is $IC(S(c_{ij}))$.

(b) We also need to tabulate the least-cost insertion that can be used to generate the error symbol locally (meaning using <u>local</u> right context only). This insertion is a solution to

$$\min_{y \in V_t^*} \{ IC(y) \mid \delta \stackrel{*}{\Longrightarrow} ya\ldots \text{ and } \delta \in lc(I_k, s)\}$$

Call this string $T(I_k, a)$. $T(I_k, a) = ?$ means no insertion is possible.

<u>Example 2.4.1</u> : Using Example 2.3.1 (Figure 2.3.3), we have $T(I_4, ")")$ = "+(a" using path (I_4, I_3, I_2), getting ϵ from (I_4, I_3) and "+(a" as "+" <u>cat</u> E(T, ")") from (I_3, I_2). ☒

The following procedure computes $T(I_k, a)$ for all $I_k \in$ closure(s) and all $a \in \hat{V}_t$:

```
procedure LocalCorrection(s);
begin
  for all I_k ∈ closure(s), all a ∈ V̂_t do
      T(I_k, a) := ? ;
      for m, n from 1 to |s| do

      (* let s_km be a least-cost terminal string obtained
         when following a path from I_k to I_m (from min path
         algorithm), if one exists, otherwise "?" .  Also,
         c_mn = "?" if no arc from m to n exists
      *)

         if IC(s_km cat Insert(c_mn, a)) < IC(T(I_k, a))
            then T(I_k, a) := s_km cat Insert(c_mn, a)
         fi
      od
  od
end LocalCorrection.
```

Figure 2.4.1 : procedure LocalCorrection

Error Corrector Procedure

Assume stack restoration is done; LR_Insert will compute a least-cost insertion string corresponding to error symbol $a \in \hat{V}_t$. Before we exhibit the procedure let us introduce the notion of an error correction graph. Let $\sigma = s_{\emptyset} \ldots s_p$ be the restored parse stack. We process the stack in a top-down fashion and, as we visit the states, we create stages (Figure 2.4.3). A stage has the error correction table of the corresponding state s associated with it. It also has one node labeled LC_i for each item in basis(s). The contents of LC_i is a string in $\hat{V}_t^* \cup \{?\}$ that is a least-cost completor (as defined below) of item I_i in state s.

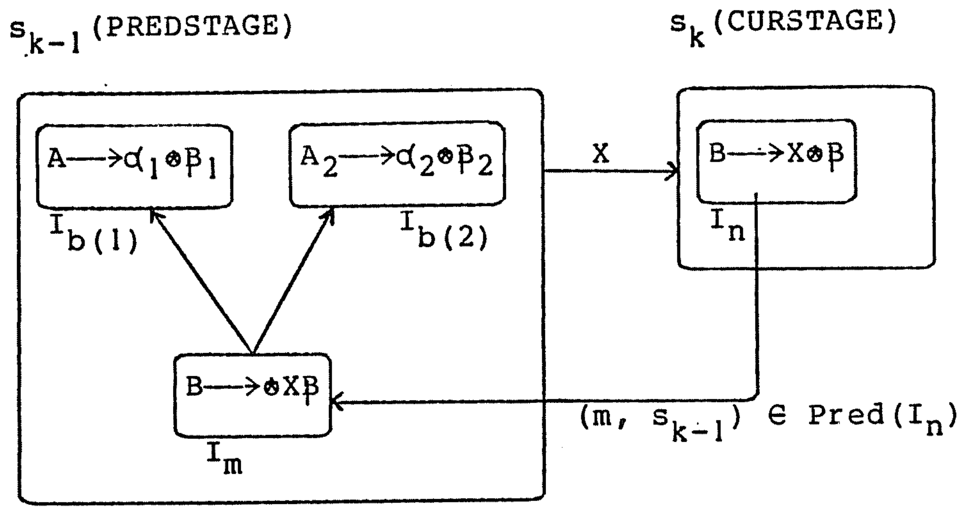
<u>Definition 2.4.1</u> : Let $\sigma = s_\emptyset \ldots s_p$ be a parse stack corresponding to some viable prefix. Assume $I = [A \longrightarrow \alpha \bullet \beta] \in s_i$ $(\emptyset \leq i \leq p)$. Then $w \in \hat{V}_t^*$ is a <u>completor</u> of I in s_i if and only if the parser when restarted in some configuration (σ, wx) can consume w and reach configuration $(s_\emptyset \ldots s_i \ldots s_j, x)$ where $s_j = GOTO(s_i, \beta)$. ⊠

Informally, a completor can be used to complete the recognition of some item in a state in the parse stack. A <u>least-cost</u> completor for I in state s is one such that there exists no completor for I that has a lower insertion cost. LC_i is maintained to cover the possibility that the error symbol will be generated by a predecessor of I_i (in a deeper stack state).

Consider Figure 2.4.2; item $I_n \in basis(s_k)$ is linked to $I_m \in s_{k-1}$, the item that produced I_n by a shift operation. This item is known to be uniquely determined by I_n and s_{k-1}. It is given by Pred(I_n). Now I_m is linked to $I_{b(1)},\ldots,I_{b(v)} \in$ basis(s_{k-1}) (where $v = v(m) \geq 1$). These are basis items of s_{k-1} reachable from I_m such that $(I_m,\ldots,I_{b(1)}), \ldots,(I_m,\ldots I_{b(v)})$ are minimal cost paths in Cl(s_{k-1}) and are given to us by the backpointer list B(I_m).

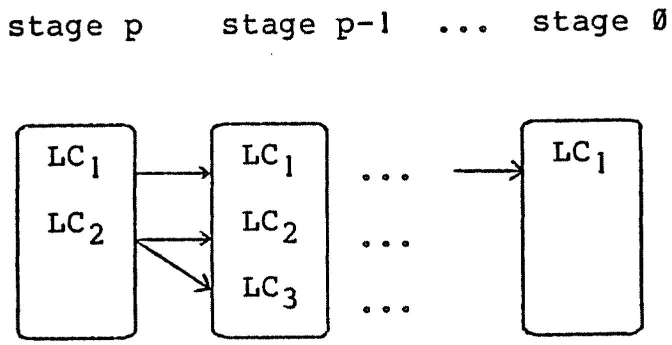
Assume $B(I_m) = \{(b(1),y_1),\ldots,(b(v),y_v)\}$. A possible value for $LC_{b(i)}$ (in the stage corresponding to s_{k-1}) is LC_n cat y_i. This value follows from the fact that we know LC_n is the lowest cost insertion necessary to complete I_m once parsing is restarted. Further, we know y_i is the lowest cost terminal string which links I_m to $I_{b(i)}$ (that is, which completes $I_{b(i)}$ given that I_m is completed). We therefore assign LC_n cat y_i to $LC_{b(i)}$ if no lower cost insertion string is known (which might complete $I_{b(i)}$ through a different closure item).

This calculation of LC values corresponds to lines 14-24 of LR_Insert. If $I_m \in$ basis(s_{k-1}), we just transfer the LC_n contents to LC_m if no lower cost insertion string is already known for LC_m (lines 35-36).



<u>**Figure 2.4.2**</u> : back-linking items

As we process the parse stack we don't need to keep track of all stages at any given time, just the current stage CURSTAGE and the previous stage PREDSTAGE are considered. STAGE(s) is a function that returns a pointer to a new stage corresponding to state s and initializes all LC_i's to "?". The following figure shows how stages of an error correction graph are processed.



<u>Figure 2.4.3</u> : error correction graph

We also want to keep track of INSERT, a least-cost insertion string that can be obtained given that the error symbol is generated by local context in a state already examined. Starting with state s_p, we initialize INSERT to an optimal solution to

$$\min \{ IC(Insert(\beta_i, a)) \mid I_i = [A_i \longrightarrow \alpha_i \bullet \beta_i] \in basis(s_p) \}$$

This calculation corresponds to lines 3-8 of LR_Insert.

Now reconsider Figure 2.4.2; if $I_n \in s_k$ is linked back to $I_m \in closure(s_{k-1})$, we want to set INSERT equal to a string that minimizes

$$\min \{ IC(INSERT), IC(LC_n \ \underline{cat}\ T(I_m, a)) \mid (m, s_{k-1}) \in Pred(I_n) \text{ and } I_m \in closure(s_{k-1}) \}$$

$$I_n \in basis(s_k)$$

That is, we consider the possibility of obtaining a lower cost value for INSERT by allowing the error symbol, a, to be generated from the local right context of I_m in s_{k-1}. $T(I_m, a)$ yields the lowest cost insertion needed to generate "a" from I_m's local right context; LC_n is the lowest cost insertion possible which will complete I_m. This computation corresponds to lines 27-33 of LR_Insert.

In general we do not need to process the stack down to stage 1. If we are processing stage k and we have $IC(INSERT) \leq IC(LC_i)$ for all i's, we know that INSERT has the optimality property we are looking for, since $IC(T(I_m, a)) > \emptyset$. This fact dictates the termination condition of the while loop in lines 11-41 of LR_Insert.

```
function LR Insert(σ, a) : TerminalString;
   σ = s_ø...s_p, the (restored) parse stack;
   a ∈ V̂_t, the error symbol;
begin
  (* initialization, using top-state info *)
  k := p; INSERT := ? ; CURSTAGE := STAGE(s_p);
  for all i such that [A_i ⟶ α_i•β_i] ∈ basis(s_p) do
     CURSTAGE.LC_i := S(β_i);
     if IC(Insert(β_i, a)) < IC(INSERT)
        then INSERT := Insert(β_i, a)
     fi
  od;
  (* now process stack, until no lower
     cost INSERT possible             *)
  while ∃ i such that
          IC(CURSTAGE.LC_i) < IC(INSERT) and k ≥ 1 do
     PREDSTAGE := STAGE(s_k-1);
     for all I_n basis(s_k) such that
             IC(CURSTAGE.LC_n) < IC(INSERT) do
       (* link I_n to predecessors in basis(s_k-1) *)
        let m be such that (m, s_k-1) ∈ Pred(I_n);
        if I_m ∈ closure(s_k-1)
           then (* follow back-ptrs to basis items *)
              for all (b(i),y_i) ∈ B(I_m) do
                 if IC(CURSTAGE.LC_n) + IC(y_i)
                    < IC(PREDSTAGE.LC_b(i))
                    then PREDSTAGE.LC_b(i)
                           := CURSTAGE.LC_n cat y_i
                 fi
              od
```

```
            (* if lower cost INSERT can be
              obtained, update INSERT *)
              if IC(CURSTAGE.LC_n) + IC(T(I_m, a))
                 < IC(INSERT)
                 then
                    INSERT := CURSTAGE.LC_n cat T(I_m, a)
              fi
           else (* we have I_m ∈ basis(s_{k-1}) *)
              if IC(CURSTAGE.LC_n) < IC(PREDSTAGE.LC_m)
                 then PREDSTAGE.LC_m := CURSTAGE.LC_n
              fi
        fi
     od;
     CURSTAGE := PREDSTAGE; k := k - 1
  end while;
  return ( INSERT )
end LR_Insert.
```

Figure 2.4.4 : function LR_Insert

LR_Insert may be used in the case G is not insert-correctable. In this case it may return "?", meaning there is no possible insertion and have to announce failure (or invoke a heuristic routine).

Example 2.4.2 : Now reconsider grammar G_2 given in Example 2.3.1 .

(1) Assuming all terminal insertion costs are set to one, we

get the following error correction tables (the notation y:a where $a \in \hat{V}_t$ and $y \in \hat{V}_t^*$ means y is to be inserted to the left of "a").

state $s_\emptyset$

basis item I_1

Pred(I_1) = ∅ (because $s_\emptyset$ is the initial state)

Insert(E$) = [a:$, ∈:a, a:+, ∈:(, (a:)]

S(E$) = "a$"

closure items I_2, I_3, I_4, I_5

B(I_k) = {(1, $)}, k = 2 to 5

T(I_k) = [∈:$, +:a, ∈:+, +:(, +(a:)], k =2 to 5

(since lc(I_k, $s_\emptyset$) = $\{+T\}^*$ cat {$}, k = 2 to 5)

state s_1

basis item I_1

Pred(I_1) = {(5, ∅), (5, 1)}

Insert("E)") = [?:$, ∈:a, a:+, ∈:(, a:)]

S("E)") = "a)"

closure items I_2, I_3, I_4, I_5

B(I_k) = {(1, ")")}, k = 2 to 5

T(I_k) = [?:$, +:a, ∈:+, +:(, ∈:)], k = 2 to 5

(since lc(I_k, s_1) = $\{+T\}^*$ cat {)}, k = 2 to 5)

state s_2

basis item I_1

Pred(I_1) = {(1, Ø)}

Insert(\$) = [∈:\$, ?:a, ?:+, ?:(, ?:)]

S(\$) = \$

basis item I_2

Pred(I_2) = {(2, Ø)}

Insert(+T) = [?:\$, +:a, ∈:+,+:(, +(a:)]

S(+T) = "+a"

<u>state</u> $\underline{s}_3$

basis item I_1

Pred(I_1) = {(4, Ø)}

Insert(∈) = [?:\$, ?:a, ?:+, ?:(, ?:)]

S(∈) = ∈

(2) Assume an error-correcting LR(1)-based parser processes "\$((\$". A syntax error is detected in configuration ($s_Ø s_1 s_1$, \$). Stack restoration leaves the configuration unchanged. LR_Insert produces the following error correction graph.

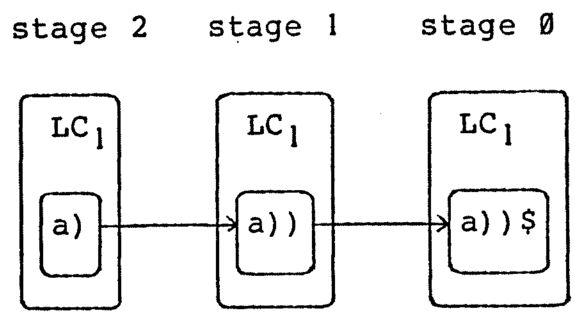


Figure 2.4.5 : error correction graph

This graph is obtained in the following way

(a) create stage 2

LC_1 := S("E)") = "a)" (line 4) using $[T \longrightarrow (\bullet E)]$ in basis(s_1). INSERT = ? since "E)" $\overset{*}{\not\Rightarrow}$ ...$... .

(b) create stage 1

We have (5, 1) ∈ Pred(I_1) in state 1 and $B(I_5)$ = {(1, ")")} in state 1 so that we link LC_1 in stage 2 to LC_1 in stage 1 and set LC_1 in stage 1 equal to "a))" (lines 23-24). No other path exists. There is no local correction that can be obtained from I_5 in stage 1 since $T(I_5$, $) = ? in state 1.

(c) create stage Ø

We have (5, Ø) ∈ Pred(I_1) in state 1 and $B(I_5)$ = {(1, $)} in state Ø so that we link LC_1 in stage 1 to LC_1 in stage Ø and set LC_1 in stage Ø equal to "a))$" (line 24). No other possibility can be found. Also $T(I_5, \$) = \epsilon$ in state 1, so that we obtain a correction by concatenating "a))" and ϵ (line 32):

INSERT := "a))"

We now exit the while loop (line 11) because we have reached the bottom of the parse stack and we finally have corrected "\$(($" into "\$((a))$".

In many cases LR_Insert computes least-cost corrections simply (and quickly) by considering only the top state on the parse stack. For example, assume an input of "\$aa$". When an error is detected, we are in configuration $(s_\emptyset s_2, a\$)$. Considering s_2 we obtain $LC_1 = S(\$) = \$$ and $LC_2 = S(+T) = +a$. Further, INSERT = Insert(+T, a) = "+". Since IC(+) is less than both $IC(LC_1)$ and $IC(LC_2)$, the computation immediately terminates (line 11) with a correction of "\$aa$" into "\$a+a$". The error corrector thus attempts to find corrections using local context in the top stack state. When necessary, however, it considers just enough states to guarantee that the lowest cost correction possible is calculated. ⊠

2.5 Properties of the Error Corrector

We now consider some of the most important properties of the class of error-correcting parsers that we have introduced. We first prove correctness: any input string can be corrected and parsed. The following lemma establishes that LR_Insert computes LC values correctly.

<u>Lemma 2.5.1</u> : During execution of LR_Insert, if LC_j in a stage corresponding to state s_i contains a string other than ?, then LC_j is a completor for basis item I_j in state s_i.

<u>Proof</u> : We follow induction on the depth of s_i in the parse stack.

Basis step: s_i is on top of the stack. Let $I_j = [A \longrightarrow \alpha \bullet \beta]$. Then $LC_j = S(\beta)$ is trivially a completor for I_j.

Induction step: assume the Lemma true for state s_{i+1}; consider s_i immediately below it in the stack. Again let $I_j = [A \longrightarrow \alpha \bullet \beta]$. Now LC_j can be assigned a value in one of two ways. If I_j has an immediate successor in s_{i+1} then LC_j is assigned the LC value of the successor (line 36). Since this LC value is a completor for I_j's successor, it must also be a completor for I_j. Otherwise, LC_j is assigned a value LC_n cat y (lines 23-24). LC_n is a completor for a closure item I_k in s_i (because it is a completor for I_k's successor in s_{i+1}) and y is derived from $\gamma \in lc(I_k, I_j, s_i)$. y can be written as $y_1 \ldots y_m$ and γ as $\gamma_1 \ldots \gamma_m$ where $\gamma_1, \gamma_2, \ldots$ are labels on a path $I_k, I_1, \ldots, I_j$ in $Cl(s_i)$ and $\gamma_1 \stackrel{*}{\Longrightarrow} y_1$, $\ldots$, $\gamma_m \stackrel{*}{\Longrightarrow} y_m$. It is easy to verify that LC_k cat y_1 is a completor for I_1 and thus by induction that LC_k cat y is a completor for I_j. ⊠

Lemma 2.5.2 : Assume that after reading and processing some input prefix $\$y \in \hat{V}_t^*$ an LR(1)-based parser invokes LR_Insert with error symbol "a". During the execution of LR_Insert, wherever INSERT contains any string $z \neq ?$, it is the case that $S' \stackrel{+}{\Longrightarrow} \$yza\ldots$.

Proof : INSERT is assigned a value in only two places and only when the new value has a cost less than the current value (and thus a cost < IC(?)). In line 6, Insert(β, a) is assigned to INSERT if the top stack state contains an item $[A \longrightarrow \alpha \bullet \beta]$. In this case the desired result follows from the definition of Insert. In line 32, an item $I_m \in closure(s_{k-1})$ is considered and INSERT is assigned a value of the form u cat t. u is the LC value corresponding to I_m's successor in s_k. By the previous Lemma, it is a completor for this item and thus also for I_m. t is equal to

$T(I_m, a)$ and may be written as t_1t_2. t_1 is derived from a path of length $\geq \emptyset$ from I_m to some item $I_n = [B \longrightarrow \bullet\gamma]$ in $Cl(s_{k-1})$. t_2 is equal to Insert(δ, a) where $I_p = [C \longrightarrow \rho\bullet B\delta]$ is an immediate successor of I_n. By an induction on path length it can be established that u cat t_1 is a completor for I_n. Thus after reading u cat t_1 the parser can reach a configuration in which the top stack state contains an item $[C \longrightarrow \rho B\bullet\delta]$ and t_2a can clearly be read from this configuration. ☒

<u>Theorem 2.5.1</u> : Assume that for some insert-correctable cfg, G, \$x... $\in$ L(G) but \$xa... $\notin$ L(G) for $x \in V_t^*$, $a \in \hat{V}_t$. Further assume that while attempting to parse \$xa... an LR parser invokes LR_Insert as soon as "a" is encountered. Then LR_Insert will return $y \in V_t^+$ such that y is an optimal solution to

$$\min_{y \in V_t^+} \{\ IC(y) \mid S' \stackrel{+}{\Longrightarrow} \$xya...\ \}$$

<u>Proof</u> : Since G is insert-correctable, some least-cost insertion string y must exist. By Lemma 2.5.2, we know any string assigned to INSERT is correct and a new value is assigned to INSERT only if it is of a lower cost than the current value. We need only therefore show that at some point an attempt to assign a string of cost IC(y) must be

made. This will be done by showing how the execution of LR_Insert traces the various ways "ya" may be recognized once parsing is restarted.

Initial step: it may be that ya... is generated by the trailing part of some basis item $[A \longrightarrow \alpha \bullet \beta]$ in the top stack state. Then it must be that $IC(Insert(\beta, a)) = IC(y)$ (since y is least-cost) and $Insert(\beta, a)$ is assigned to INSERT in this case (line 6). Otherwise write "ya" as $y_1 y_2 a$ and assume $y_1 \in V_t^*$ is used to complete some basis item $I_i = [B \longrightarrow \gamma \bullet \delta]$. y_1 must be least-cost and thus $IC(y_1) = IC(S(\delta)) = IC(LC_i)$. If $IC(INSERT) \geq IC(LC_i) = IC(y_1)$ we go on to the next step (otherwise a least-cost solution has already been found).

Iterative step: we have just completed processing a basis item I_i in state s_j where $IC(LC_i) = IC(y_1)$. We continue by tracing how $y_2 a$ might be recognized. I_k, I_i's predecessor in s_{j-1} is considered. It may be the case that $y_2 a$ is fully recognized by items in s_{j-1}. If this is so, a sequence of items I_k, $I_{m(1)}$, ... , $I_{m(n)}$ $(n \geq 1)$ in $Cl(s_{j-1})$ must exist where segments of $y_2 a$ are used to complete in turn $I_{m(1)}$, ..., $I_{m(n-1)}$ and the remainder of the string is recognized by the trailing part of $I_{m(n)}$. Now it must be the case that $IC(T(I_k, a)) = IC(y_2)$ since computation of T (in procedure LocalCorrection) considers all possible paths from an item and, by assumption, y_2 is least-cost. Thus in line 32 INSERT can be assigned a string of cost $IC(y_1)+IC(y_2) = IC(y)$.

Otherwise, write y_2a as z_1z_2a and assume $z_1 \in V_t^*$ is used to complete items in s_{j-1}. A sequence of items I_k, $I_{m(1)}$, ..., $I_{m(n)}$ $(n \geq \emptyset)$ will be followed where $I_{m(n)} \in basis(s_{j-1})$ and segments of z_1 will be used to complete, in turn $I_{m(1)}$, ..., $I_{m(n)}$. If $n = \emptyset$ then $I_k = I_{m(n)}$ and $LC_{m(n)}$ can be assigned a string of cost $IC(y_1)$ (line 36) and $z_1 = \epsilon$. If $n > \emptyset$ then $IC(z_1) = IC(v)$ where $(m(n), v) \in B(I_k)$ (since z_1 must be least-cost) and $LC_{m(n)}$ is assigned (in lines 23-24) a string of cost $IC(y_1)+IC(z_1)$. In either case $LC_{m(n)}$ cannot contain a lower cost string since, by Lemma 2.5.1, this could be used to complete $I_{m(n)}$ and a lower cost insertion than y would result. If $IC(INSERT) > IC(LC_{m(n)}) = IC(y_1)+IC(z_1)$ this step is repeated on the next state down the parse stack with $I_{m(n)}$ renamed I_i, y_1z_1 renamed y_1 and z_2a renamed y_2a. If $IC(INSERT) \leq IC(LC_{m(n)})$ the algorithm may terminate but a least-cost INSERT must already have been found since $IC(LC_{m(n)}) \leq IC(y)$.

The iterative step is repeated until the state which finishes the recognition of ya is processed or until IC(INSERT) is less or equal to the cost of all LC values.

In either case a simple induction on the number of iterative steps executed establishes that an INSERT value of cost IC(y) must be obtained. ⊠

We now analyze the efficiency of our error correcting parser. We first present a quadratic upper bound and later show conditions in which linearity can be guaranteed.

Theorem 2.5.2 : Assume an LR(1)-based parser using LR_Insert as an error corrector processes x and corrects it into x'. Then it is the case that $|x'| = O(|x|)$.

Proof : We will charge each symbol inserted during error correction to some input symbol and show that each input symbol is charged for at most a constant number of insertions.

For charging purposes we associate each state with an input symbol. Assume that during normal parsing (when the lookahead symbol is in x$), the stack height is h when symbol "a" is first used as a lookahead. Any states added by "a" at a height greater than h are charged to a; those at height $\leq$ h retain the association in effect when "a" was first used. It is easy to establish that the number of states so charged to "a" will be bounded by a constant and will not increase as parsing progresses.

Now assume LR_Insert is invoked with error symbol "b" and a stack $\sigma = s_1 \ldots s_j$. Starting with s_j, states are visited until at state s_i $(i \leq j)$, the final value of INSERT is determined. (The fact that LR_Insert may have to do some processing further down the stack to verify optimality of this correction is of no significance in this proof.)

INSERT can be written as LC cat LOCAL where LC is determined by $s_{i+1} \ldots s_j$ and LOCAL is determined by s_i (and of course "b"). The portion of LC contributed by each of s_{i+1} to s_j can be bounded in length by a constant and is charged to the input symbol associated with each such state.

By construction, LC is a completor for some closure item $[A \longrightarrow \bullet\alpha]$ in s_i (if $i < j$) and after LC is fully parsed, the A-successor to s_i is the stack top. Since $s_{i+1}, \ldots, s_j$ have been popped, they cannot be charged again for any portion of an LC string. Then LOCAL is processed. Its length can be bounded by a constant and it is charged to "b". After LOCAL is parsed and just before normal parsing is resumed with "b" as the lookahead, the number of states above s_i in the stack can be bounded by a constant (determined by s_i, A and LOCAL). Each of these states (created during error processing) is charged to "b".

We now observe that the total number of states charged to a given input symbol and used to contribute a portion of LC is bounded by a constant and thus so is the total number of LC symbols charged to that input symbol. So too, an input symbol is charged at most once for LOCAL which is of bounded length. The desired result is immediate. ⊠

Theorem 2.5.3 : Assume an LR(1)-based parser using procedure LR Insert as an error corrector processes \$x\$. Then it requires at most $O(|x|^2)$ time.

Proof : Assume first a canonical LR(1) parser is used. It is easy to establish that given a careful implementation of procedure LR_Insert, the time required to process each stack state during correction can be bounded by a constant. By Lemma 2.5.2, at most $O(|x|)$ states can be processed during any invocation of the corrector and no more than $|x|$ invocations are possible. The $O(|x|^2)$ time bound follows immediately.

In the case stack restoration is needed, procedure restore requires at most $O(|x|^2)$ additional time in all (Theorem 2.2.1). ⊠

There is a strong reason to believe that the above quadratic worst case will not be realized in practice. Certainly for common programming languages and most syntax errors, local context (derived from the upper-most stack states) will suffice. Even more important, LR(1)-based parsers that are used in practice invariably use a bounded depth parse stack. For example, the University of Wisconsin PASCAL compiler [Fis 77] uses a parse stack of maximal depth 101 that has sufficed for even the largest of programs in two years of operation. (The PASCAL compiler compiles itself and has more than 40,000 tokens.)

For this very important special case, we can establish linearity of our error correcting parser. We start by establishing linearity of stack restoration.

Lemma 2.5.3 : Assume that a bounded depth parse stack LR(1)-based parser using LR_Insert as an error corrector processes x. Then procedure restore requires at most $O(|x|)$ time in all.

Proof : Let α be the viable prefix corresponding to the parse stack just before buffering begins and let β be the viable prefix corresponding to the parse stack when the error is detected. By the correct prefix property, and the fact that no shifts occured during buffering, we have for some $d \in \hat{V}_t$ $S' \overset{+}{\underset{r}{\Rightarrow}} \beta d\ldots \overset{*}{\underset{r}{\Rightarrow}} \alpha d\ldots \overset{*}{\underset{r}{\Rightarrow}} yd\ldots$. Further since $|\alpha|$ and $|\beta|$ are bounded by a constant, so is y (if it is chosen properly). By Theorem 2.5.2, there exists $z \in \hat{V}_t^*$

such that $ydz \in L(G)$ and $|ydz|$ is bounded by a constant. The total number of moves required to parse ydz is bounded by a constant (and thus also is the total number of moves buffered in the auxiliary stack AS in reducing α to β). Therefore procedure restore requires only a constant time per invocation and at most $O(|x|)$ time in all. ⊠

<u>Theorem</u> <u>2.5.4</u> : Assume a bounded depth parse stack LR(1)-based parser using LR_Insert as an error corrector processes \$x\$. Then the processing requires at most $O(|x|)$ time and $O(|x|)$ space.

<u>Proof</u> : In the case a canonical LR(1) parser is used, linearity is immediate since one invocation of LR_Insert can process the entire (bounded depth) parse stack in constant time, using an amount of space bounded by a constant.

In the case stack restoration is needed, procedure restore requires at most $O(|x|)$ additional time and $O(|x|)$ additional space (Lemma 2.5.3). ⊠

We can also create a more general (but in practice less useful) linear-bounded error corrector. To eliminate the need for parse stack restoration, we will assume a canonical LR(1) parser (which subsumes all LR(1)-based parsing techniques). A variant of function LR_Insert that performs a bottom-up rather than top-down parse stack traversal is used to determine least-cost corrections (this function, named BU_LR_Insert, is detailed in Appendix A.3). Unlike LR_Insert, this function will always need to examine the entire parse stack to determine a correction. Nevertheless, it has the very useful property that a given parse stack state will never need to be visited more than once for a given terminal error symbol. This property results because all intermediate information characterizing the state of the error corrector while visiting some parse stack state s is determined solely by those parse stack states below s in the stack and the error symbol, a. This intermediate information can be stored in the parse stack with s (or alternately in a parallel stack). Distinct intermediate information may be stored with a given stack state for each possible terminal error symbol.

Now if we invoke the error corrector with an error symbol b, and some parse stack states, during a previous error corrector invocation, were already visited with error symbol b, these states need not be revisited. Rather, using the intermediate information previously stored, the error corrector can be started at the state just above the last state previously visited with b. Given a careful, but straightforward, implementation of these ideas the following result can be established.

Theorem 2.5.5 : Assume an LR(1) error correcting parser using an implementation of function BU_LR_Insert as described above processes \$x\$. Then it requires at most $O(|x|)$ time and $O(|x|)$ space.

Proof : Follows immediately from the fact that no parse stack state needs to be visited more than $|\hat{V}_t|$ times for error correction purposes. ☒

In summary, LR_Insert appears to be a simple and effective automatic error corrector. For those bounded depth parse stack, LR(1)-based parsers used in practice, correctness and linearity can be guaranteed. Further, for any LR(1)-based cfg an error correcting parser with a linear worst case can be created.

2.6 Testing Insert-Correctability

LR_Insert may be used with any LR(1)-based parser that is based on an insert-correctable cfg. Often we can determine insert-correctability directly from the properties of the language the cfg specifies. However, in general a decision procedure can be obtained by extending the definition of LR(Ø)-items (or LR(1)-items, for that matter). While building an extended CFSM, we consider items of the form $I = [A \longrightarrow \beta_1 \bullet \beta_2, t]$ where $t : \hat{V}_t \longrightarrow \{\underline{true}, \underline{false}\}$ is such that

$\forall a \in \hat{V}_t, \forall \alpha\beta_1$ for which I (in state s) is valid $t(a) = \underline{true}$ if and only if I (in s) is valid for some right sentential form $\alpha\beta_1 \bullet \beta_2 w$ where $w = \ldots a \ldots$. Call this condition (*).

We now outline an algorithm that computes the extended CFSM, followed by a lemma showing that the t-tables that are computed satisfy (*).

(1) The basis of the initial state is $\{[S' \longrightarrow \$\bullet S\$, t_\emptyset]\}$ where $t_\emptyset(a) = \underline{false}$ for all $a \in \hat{V}_t$.

(2) The basis of the X-successor s' of state s is obtained as follows:

```
basis(s') := ∅;
for all [A ⟶ β1•Xβ2, t] ∈ s do
   basis(s') := basis(s') u [A ⟶ β1X•β2, t]
od
```

(3) For a closure item $I = [A \longrightarrow \bullet\gamma, t_1] \in s$ we set $t_1(a) = \underline{true}$ if and only if

(a) $t(a)$ = true for any basis item I' $\in$ s which is a descendant of I in Cl(s).

or (b) ...a... $\in$ L(lc(I,s)). ⊠

Lemma 2.6.1 : (1) Let I = $[A \longrightarrow \beta_1 \bullet B\beta_2, t]$ be a basis item in state s and let $\alpha\beta_1 \bullet B\beta_2 w$ be any right sentential form for which I (in s) is valid. If J = $[C \longrightarrow \bullet\gamma, t']$ is a closure item in s then J is valid for $\alpha\beta_1 \bullet \gamma vw$ where v $\in$ L(lc(J, I ,s)).

(2) Let J = $[C \longrightarrow \bullet\gamma, t]$ be a closure item in state s and assume J (in s) is valid for $\alpha\bullet\gamma z$. Then there exists a basis item I = $[A \longrightarrow \beta_1 \bullet B\beta_2, t']$ in s such that I is valid for $\delta\beta_1 \bullet B\beta_2 w$ where $\delta\beta_1 = \alpha$ and $z = vw$ for some v $\in$ L(lc(J, s)).

Proof : Part (1) may be proved by a simple induction on path length from J to I in Cl(s).

Part (2) follows from a simple induction on path length from J to I in Cl(s), using the observation that if closure item J is valid for a right sentential form, this sentential form must have a predecessor in the derivation for which some immediate successor of J in Cl(s) is valid. ⊠

Lemma 2.6.2 : If condition (*) holds for all basis items of state s, it holds for all closure items.

Proof : Consider closure item J = [C $\longrightarrow$ $\bullet\delta$, t], any a $\in \hat{V}_t$ and any viable prefix γ such that J (in s) is valid for γ. If t(a) = true, then by construction of the extended CFSM either step (3a) or step (3b) must hold. If (3a) holds, let the basis item I' be [A $\longrightarrow$ $\beta_1 \bullet B\beta_2$, t] where $\gamma = \alpha\beta_1$. By (*) since t(a) = true, I' is valid for $\alpha\beta_1\bullet B\beta_2 w$ where w = ...a... . But by Lemma 2.6.2 (1), we know J is valid for $\alpha\beta_1\bullet\delta vw$ as required. If (3b) holds, let I' be as above and assume v = ...a... $\in$ L(lc(J, I', s)). Then for all $\alpha\beta_1\bullet B\beta_2 w$ for which I' is valid, we know (by Lemma 2.6.2 (1)) that J is valid for $\alpha\beta_1\bullet\delta vw = \gamma\bullet\delta vw$ as required by (*).

If J is valid for $\gamma\bullet\delta z$ where z = ...a... then by Lemma 2.6.2 (2), for some basis item I = [A $\longrightarrow$ $\beta_1\bullet B\beta_2$, t_1], it is the case that I is valid for $\alpha\beta_1\bullet B\beta_2 w$ where $\alpha\beta_1 = \gamma$ and z = vw for some v $\in$ L(lc(J, s)). If w = ...a... then (since (*) holds for I), t_1(a) = true and thus t(a) = true by construction. Otherwise ...a... = v $\in$ L(lc(J,s)) and again, by construction, t(a) = true. ☒

Lemma 2.6.3 : Assume we construct an extended CFSM as outlined above. Then condition (*) holds for every item in every state.

Proof : An induction on the order in which states are created: (*) trivially holds for the sole basis item of $s_\emptyset$ and by Lemma 2.6.2 then holds for all of $s_\emptyset$. In like manner (*) holds for the basis items of any state that is added because it holds for the (already existing) items from which the basis items were created (by a successsor operation). By Lemma 2.6.2, (*) then holds for all items of the newly created state. ☒

Definition 2.6.1 : A state s of the extended CFSM corresponding to an augmented LR(1) grammar G is safe if and only if for all $a \in \hat{V}_t$ there exists a basis item $I = [A \longrightarrow \beta_1 \bullet \beta_2, t]$ in s such that $\beta_2 \overset{*}{\Longrightarrow} \ldots a \ldots$ or $t(a) = \underline{true}$.

Theorem 2.6.1 : An augmented LR(1) grammar G is insert-correctable if and only if all states of the extended CFSM corresponding to G are safe.

Proof : (If part) Assume x has been read and reduced to viable prefix γ. Assume further we are in state s. Since s is safe, for any $a \in \hat{V}_t$ there exists a basis item $I = [A \longrightarrow \beta_1 \bullet \beta_2, t]$ for which $\beta_2 \overset{*}{\Longrightarrow} ya\ldots$ or $t(a) = \underline{true}$. Clearly for the former case $S' \overset{+}{\Longrightarrow} xya\ldots$. In the latter case, by (*) $S' \underset{r}{\overset{+}{\Longrightarrow}} \alpha\beta_1\beta_2 w$ where $w = \ldots a \ldots$ and $\gamma = \alpha\beta_1$. Thus $S' \overset{+}{\Longrightarrow} x \ldots w = x \ldots a \ldots$.

(Only if part) Now assume s is not safe and assume the items in s are valid for δ where $\delta \overset{*}{\Longrightarrow} x \in \hat{V}_t^*$. Choose any basis item $I = [A \longrightarrow \beta_1 \bullet \beta_2, t]$ in state s. Since s is not safe $\beta_2 \overset{*}{\not\Longrightarrow} \ldots a \ldots$. Further by (*) no sentential form $\alpha\beta_1\bullet\beta_2 w$ in which $w = \ldots a \ldots$ can exist (otherwise $t(a)$ = true). Thus item I cannot participate in any parse which will eventually allow "a" to be accepted. But neither can any other basis item, so G cannot be insert-correctable. [X]

Example 2.6.1 : Consider the following LR(1) grammar G_3

$S' \longrightarrow \$S\$$

$S \longrightarrow (S) \mid a$

Figure 2.6.1 shows part of the extended CFSM for G_3, indicating t-table values for the terminal symbol ")" only.

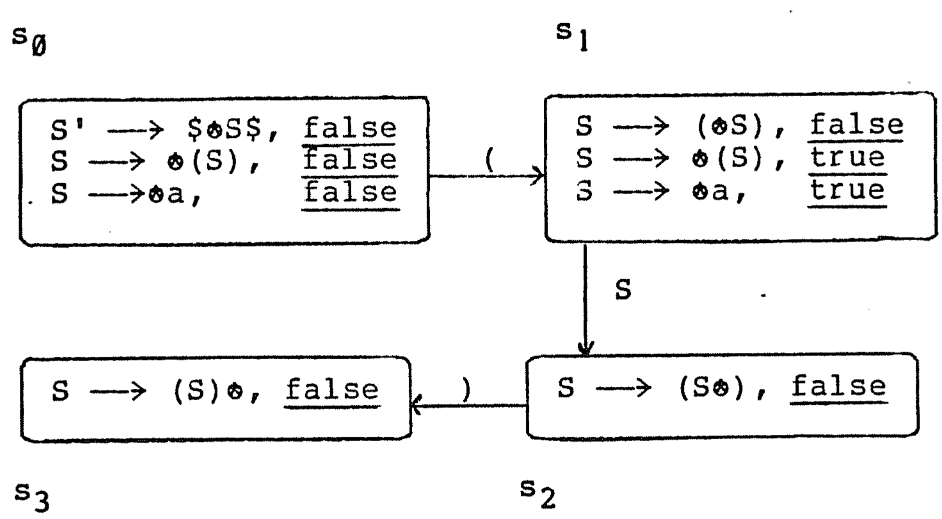


Figure 2.6.1 : extended CFSM for G_3

The reader may easily verify that states $s_\emptyset, s_1$ and s_2 are safe w.r.t. ")" and that state s_3 is not safe. Therefore grammar G_3 is not insert-correctable. For example, an input of $(a))$ will cause LR_Insert to fail on the second ")". ⊠

Finally, it should be noted that the class of insert-correctable LR languages is large and interesting. Surprisingly enough, a language such as ALGOL 60 is insert-correctable after a very minor modification: one has to allow a program to be a sequence of blocks rather than a single block [FMQ 77]. Otherwise, LR_Insert would fail on an input string of the form "begin ... end end" (for the same reason as in the above example).

Concatenation of program segments to form a larger program segment is a very common and almost universal means of building programs. If we allow whole programs to be concatenated to form valid programs, then insert-correctability is immediate.

Chapter 3 : A LOCALLY LEAST-COST ERROR CORRECTOR FOR LR(1)-BASED PARSERS

In this chapter we extend the error corrector of Chapter 2 to accomodate deletions as well as insertions. The error corrector we develop corresponds to the model of Definition 1.3.3 . It is able to correct any LR(1)-based cfg.

3.1 The Error Corrector

Assume the input string $z = \$xa_1 \ldots a_n\$$ is such that $\$x\ldots \in L(G)$ but $\$xa_1\ldots \notin L(G)$. We are looking for a solution (i, y) to †

$$\min_{0 \le i' \le n} \{ \min_{y' \in V_t^*} \{ DC(a_1 \ldots a_{i'}) + IC(y') \mid \$xy'a_{i'+1} \ldots \in L(G) \} \} \quad (*)$$

† This problem (*) was previously stated in Definition 1.3.3 .

This minimization model could be implemented very easily by calling LR_Insert repeatedly with error symbols a_1, a_2, ... until we reach a situation where the cumulative deletion cost $DC(a_1 \ldots a_{i+1})$ is larger than $IC(y)+DC(a_1 \ldots a_i)$ where y is the insertion string computed by LR_Insert(σ, a_i). However, this procedure has a $O(|z|^2)$ worst case running time because during each of $O(|z|)$ possible invocations of the error corrector, we may need to examine all of the remaining input symbols to determine a locally least-cost correction.

Now assume that, upon detection of the first syntax error, we do the following preprocessing of the remaining input string $a_1 \ldots a_n\$$. We maintain a vector

First_Occurence : $\hat{V}_t \longrightarrow \{1, \ldots, n+1\} \cup \{\underline{absent}\}$

where First_Occurence(a) points to the first occurence of "a" in $a_1 \ldots a_n\$$, if any. Also, each position in the input string $a_1 \ldots a_n\$$ is labeled with D such that $D(i) = DC(a_1 \ldots a_{i-1})$ for $1 \leq i \leq n+1$, $D(\underline{absent}) = \infty$ and $D(\emptyset) = \emptyset$. We now consider $c \in \hat{V}_t$ which is a solution to

$$\min_{c' \in \hat{V}_t} \{ D(\text{First\_Occurence}(c'))+IC(\text{LR\_Insert}(\sigma, c')) \} \qquad (**)$$

Problems (*) and (**) are equivalent in the sense that their solutions correspond to the same correction values (i.e., c = a_{i+1} and y = LR_Insert(σ, c)). This is because, if we delete input symbols up to symbol b, then we need only delete up to the first occurence of b in the remaining input string (deleting up to a later occurence cannot lead to a lower cost insertion). This second problem (**) was first introduced by Fischer and Milton [FM 77] for the modified FMQ LL(1) corrector. The following procedure LR_Corrector gives a straightforward solution to problem (**). Upon return from the procedure, w is the corrected remaining input string.

```
procedure LR_Corrector(σ, w) ;
   σ, the parse stack;
   w = a1...an$, the (preprocessed) remaining input string;
begin
   restore(σ); (* if necessary *)
   y := ? ; i := 0 ;
   for all c ∈ V̂t do
      if First_Occurence(c) ≠ absent then
         j := First_Occurence(c);
         z := LR_Insert(σ, c);
         if D(j)+IC(z) < D(i)+IC(y)
            then y := z; i := j
         fi
      fi
   od;
   w := yai...an$
   (* that is, delete a1...ai-1 and then insert y *)
end LR_Corrector.
```

Figure 3.1.1 : procedure LR_Corrector

Since we have the correct prefix property, we can guarantee some correction for which $y \in V_t^+$ or $1 \leq i \leq n$ (or both) will be found. By construction the y and i chosen will define a locally least-cost correction. Also note that in order to maintain First_Occurence efficiently we need to link every preprocessed symbol in the remaining input string to its next occurence.

These links are used to update First_Occurence when symbols are read and/or deleted. The details of this process are left to the interested reader.

3.2 Properties of the Error Corrector

The following theorems summarize the properties of LR_Corrector.

Theorem 3.2.1 : Assume that for some cfg G and some input string $z = \$xa_1 \ldots a_n\$$, $\$x\ldots \in L(G)$ but $\$xa_1\ldots \notin L(G)$. Further assume that while attempting to parse z an LR(1)-based parser invokes LR_Corrector as soon as a_1 is encountered. Then LR_Corrector will choose correction values y and i as specified by problem (*).

Proof : Follows immediately from the equivalence of problems (*) and (**) and the correctness of LR_Insert (Theorem 2.5.1). ⊠

Theorem 3.2.2 : Assume an LR(1)-based parser using LR_Corrector as an error corrector processes \$x\$. Then it requires

(1) at most $O(|x|^2)$ time and $O(|x|)$ space in the gen-

eral case.

(2) at most O(|x|) time and space if a bounded depth parse stack is assumed.

(3) at most O(|x|) time and space if a canonical LR(1) parser is assumed.

Proof : We first note that the preprocessing of the input string (i.e. the computation of First_Occurence and D) takes linear time and space. We now consider different cases:

(1) In the general case, for each of O(|x|) possible errors, it may take O(|x|) time to restore the parse stack (Theorem 2.2.1) and O(|x|) time for every invocation of LR_Insert (Theorem 2.5.3). Thus we obtain the desired result.

(2) In the case of a bounded depth parse stack, stack restoration takes O(|x|) time in all (Theorem 2.5.3) and every invocation of LR_Insert takes constant time (Theorem 2.5.4). Thus we obtain the desired result.

(3) When a canonical LR(1) parser is used, stack restoration is not needed. Moreover, as discussed in the previous chapter (Theorem 2.5.5), we can guarantee linearity of LR_Insert by using the bottom-up stack traversal error corrector. ⊠

Every invocation of LR_Corrector may require up to $|\hat{V}_t|$ invocations of LR_Insert (line 6). In practice, one could do the preprocessing of the remaining input string incrementally. That is, one would first compute the cost of corrections involving 0 deletions, then 1 deletion, etc... , calling LR_Insert at most once for a given terminal symbol. As soon as the best known correction is no more expensive than the cumulative deletion cost, processing can be terminated. Since deletion costs are often chosen to be rather large (to discourage wholesale deletion of user programs), we normally expect this incremental approach to be very effective.

3.3 Implementation Results

The error corrector described above has been implemented in SIMULA 67 on a UNIVAC 1110 computer. It consists of two programs: an LALR(1) constructor that builds the parsing table and the error correction tables, and an LR(1)-based parser using an implementation of LR_Corrector as an error corrector.

Error correction tables are built as described in Chapter 2, with the exception that T values are not explicitly computed by the table generator. Rather, closure graphs (CL(s)) are tabulated. Computation of a T value is done, as needed, by LR_Insert. However, once a T value is computed it is saved and need not be recomputed. Using this method, we can provide faster correction for common syntax errors while keeping the size of the error correction tables reasonably small. Error correction tables, which are used on an exception basis, can be kept in secondary storage. The algorithm therefore operates quite efficiently with a rather small primary storage requirement.

The error corrector was tested on an LALR(1) grammar for PASCAL † . Table computations (for both parser and error corrector) require 6 minutes and 40 seconds on the UNIVAC 1110 for a grammar with 53 terminals, 89 terminals and 195 productions (the CFSM having 182 states). The total size of the error correction tables is 130K words in the case the T table file is empty. (This size includes S and E tables and closure graphs represention.) Although not negligeable, this size is not beyond the capability of common secondary storage.

---

† The cfg that is suggested in [JW 75] had to be modified to remove some ambiguities.

The problem of assigning insertion and deletion costs is subject to some heuristic considerations. The cost functions used for PASCAL are given in Appendix A.4. Deletion costs have been set higher than insertion costs. In this way we encourage corrections that build upon the existing input strings. This weighting also allows greater efficiency of the correction process. For insertion costs, we assigned a higher cost to symbols that announce a complex syntactic structure (e.g., "if", "[", etc...) as opposed to symbols terminating such a structure (e.g., ";", "]", etc...). Test cases were used for tuning correction costs.

The following program provides examples of the kind of corrections effected by LR_Corrector. This example has been previously presented by Graham and Rhodes [GR 75], Tai [Tai 78], Poplawski [Pop 78] and Penello and DeRemer [PD 78] to illustrate their respective methods as well as the operation of the Cornell PL/C compiler [CW 73] and the Zurich PASCAL compiler [JW 75].

Example 3.3.1 : We first present the input program itself. A "↑" is used to mark symbols considered erroneous. This listing would correspond to the output listing in the case LR_Corrector is used for the sole purpose of error recovery. Next, the corrections effected by LR_Corrector are included. Insertions are underlined by *'s and deletions are commented out by {}.

```
program example(input, output);
var
  a, b : array[1..5  1..10] of integer;
                     ↑
  i, j, k, l : integer;
begin
  3: i + j > k + l * 4
       ↑
       then go 2
       ↑       ↑
       else k is 2 ;
              ↑  ↑ ↑
  a  1, 2  := b[3*(i+4 , j* /k ]
     ↑↑    ↑           ↑    ↑
  if i = 1 then then goto 3 ;
  ↑             ↑
2: end.
```

```
program example(input, output);
var
  a, b : array[1..5 , 1..10] of integer;
                    *
  i, j, k, l : integer;
begin
  3: i := + j > k + 1 * 4 ;
       **                 *
  if CONSTANT then go := 2
  ***********         **
              else k := is[2] ;
                     **   * *
  a := 1 {,} + 2 ; ID := b[3*(i+4) , j*CONSTANT/k ] ;
    **       *   *******         *     ********     *
  if i = 1 then if CONSTANT then goto 3;
                ***********
2: end.
```

Figure 3.3.1 : PASCAL test program

First considering the error recovery aspect, we can see that 14 errors are detected and the position of the ↑-markers allows for prompt correction of all errors by a knowledgeable programmer. The cascading effect is fairly limited.

Now considering the error correction aspect, we can see that most of the corrections effected by LR_Corrector are quite reasonable. However some problems do arise. For example, it is most likely that "a[1,2] := b..." was intended in line 9. This correction is indeed the choice made by other correctors using a "forward move" algorithm [GR 75], [PD 78], where part of the remaining input string is parsed before correction (in this case "1,2" appears to be a subscript list rather than an expression). Instead of parsing ahead, which can have some undesirable effects on the overall translation process, we propose another approach to solve this problem. In the given context, "a := 1..." is illegal (in a context-sensitive sense) since "a" is an array. Therefore "[" is the only legal insertion. Although the techniques of Chapters 2 and 3 do not allow for such considerations, since they apply solely to context-free parsing, we will show in the next chapter how we can take them into account. There we develop a locally least-cost corrector for a context-sensitive parser.

Another problem occurs in line 6 where the insertion of "if" immediately after "3:" would be preferred. In fact, other correctors make this choice rather than ours. However they obtain this correction by using a "backward move" [GR 75], where modification of the left-context is considered. Following Fischer et al. [FMQ 77] and Watt [Wat 76], we find backward moves highly undesirable in a one-pass compiler where input symbols have to be accepted at some point so that they can be used for translation purposes. Moreover, it has been noted by Watt that most syntax errors can be satisfactorily corrected by transformation of the remaining input string only. ☒

In summary, the LR corrector that we have presented above has both theoretical and practical significance. Theoretically, the algorithm can be shown to operate correctly on <u>any</u> input string. A least-cost correction is guaranteed and in cases of special interest (bounded depth parse stack), linearity can be established. On the practical side, preliminary experience indicates that our corrector can be used satisfactorily with most LR-driven compilers. In particular it allows an error recovery or error correction capability to be added automatically with little, if any, impact on the overall structure of the compiler.

As noted above, our definition of a least-cost correction is a very <u>local</u> one since it is concerned with finding an insertion that allows the first non-deleted input symbol to be accepted by the parser. In some cases, other methods obtain more plausible corrections by using more <u>global</u> schemes. We will present results that suggest that comparable (and in many cases superior) corrections can be obtained if a local minimization model can include context-sensitive information in the correction process.

Chapter 4 : CONTEXT-SENSITIVE ERROR CORRECTION

4.1 Introduction

Context-free grammars permit the specification of the context-free syntax of programming languages. This specification (BNF) can be used to generate efficient parsers. However, not all aspects of the syntax of programming languages are of a context-free nature (e.g., correspondance between declaration and usage of identifiers in PASCAL). Traditionally, such context-sensitive restrictions have been stated informally in (for example) English. During compilation they are enforced by hand-coded semantic routines that are invoked by the context-free parser.

Attributed grammars were introduced by Knuth [Knu 68] as a simple mechanism for extending context-free grammars to include context-sensitive information. Informally, each grammar symbol posesses a set of attribute positions. For example, a terminal CONSTANT might have two attribute positions: one for its type, one for its value. Also, attribute evaluation rules are associated with context-free productions. As shown by Lewis et al. [LRS 76], Watt [Wat 77a], Milton [Mil 77] and others, attributed grammars can be used to generate context-sensitive parsers automatically. This construction has the advantage of providing the compiler writer with a way of specifying the flow of context-sensitive information in a non-procedural manner.

As a natural extension to the work of the previous chapters, we now explore the possibility of generating locally least-cost error correctors for context-sensitive parsers. The possibility of using context-sensitive information in choosing corrections was mentioned by Feyock and Lazarus [FL 76]. However, they did not present any formal way of making the context available to the error corrector. When an error is detected, there is a wealth of information available in the values of the attributes. For example, in the case of an undefined identifier, the entire symbol table is available. In order to make use of this information we will incorporate attributes into the error correction process.

4.2 Attributed Grammars

We first define attributed grammars. Different formalisms have been presented to specify how attribute values are to be evaluated. The definition we present uses action functions to specify evaluation rules other than simple transfers of attribute values. It is very similar to the definitions given in [Wat 77a] and [LRS 76].

Definition 4.2.1 : An attributed grammar (ag) is a 10-tuple AG = (V_n, V_t, Q, S, A, AD, R, IS, P, F_Q) where

V_n is a finite set of nonterminal symbols.

V_t is a finite set of terminal symbols, disjoint from V_n.

Q is a finite set of primitive predicate symbols, disjoint from $V_n \cup V_t$.

S is a distinguished element of V_n, the start symbol; it does not appear on the right-hand side of any production in P.

A is a finite set of attribute variables.

AD is a finite set of attribute domains.

R is a mapping from A to AD, the range function.

IS is the control of AG, a collection of 4-tuples IS_x = (M_x, N_x, i(x), s(x)) for each $x \in V_n \cup V_t \cup Q$. $M_x \geq 0$ is the number of inherited attribute positions of x, $N_x \geq 0$ is the number of synthesized attribute positions of x, i(x) is an M_x-tuple of attribute domains in AD which are the domains of the inherited attribute positions of x and s(x) is an N_x-tuple of attribute domains in AD which are the domains of the synthesized attribute positions of x. For each $x \in V_t$, we require $M_x = 0$ (that is, terminal symbols do not have inherited attribute positions).

P is a finite set of productions of the form

$$A{\downarrow}a_1^0 \ldots {\downarrow}a_{M_0}^0 {\uparrow}b_1^0 \ldots {\uparrow}b_{N_0}^0$$

$$\longrightarrow U_1{\downarrow}a_1^1 \ldots {\downarrow}a_{M_1}^1 {\uparrow}b_1^1 \ldots {\uparrow}b_{N_1}^1$$

$$\vdots$$

$$U_m{\downarrow}a_1^m \ldots {\downarrow}a_{M_m}^m {\uparrow}b_1^m \ldots {\uparrow}b_{N_m}^m$$

where $A \in V_n$, $M_0 = M_A$, $N_0 = N_A$ and $U_k \in V_n \cup V_t \cup Q$, $M_k = M_{U_k}$, $N_k = N_{U_k}$ for $k = 1,\ldots,m$. Inherited attribute positions are prefixed by "↓", synthesized positions by "↑". Each a_k^j or b_k^j is either an attribute variable or a constant attribute value. (The a_k^j's and b_k^j's will be used to specify how attribute values are to be assigned to attribute positions.)

F_Q is a finite set of <u>action functions</u>. For each $X \in Q$ there exists $f_X \in F_Q$ such that

$$f_X : i(X) \longrightarrow s(X) \times \{\underline{true}, \underline{false}\}.$$

f_X is total recursive over $i(X)$. ⊠

Attributed grammars will be augmented in the same way as context-free grammars. "$" is a terminal symbol which does not have any attribute positions.

We now explain how the evaluation of attributes is specified by an ag. Informally, the definition of attribute values is specified by the use of attribute variables and constant attribute values. The role of a primitive predicate is twofold. Given values for its inherited attribute positions, it evaluates its synthesized attribute positions. It can also be used to perform checks on the validity of the application of a production. Whenever it returns <u>false</u>, the presence of a context sensitive error is detected. This corresponds to an "illegal" application of a production and thus blocks a derivation under the rules of the ag.

Considering the application of a production, we distinguish two kinds of attribute positions: a <u>defining</u> position that is used as a source in copying an attribute value and an <u>applied</u> position which is used as a sink in copying an attribute value.

Definition 4.2.2 : A defining attribute position is an inherited attribute position of the left-hand side of a production or a synthesized attribute position of a symbol on the right-hand side of a production. An applied attribute position is a synthesized attribute position of the left-hand side of a production or an inherited attribute position of a symbol on the right-hand side of a production. ⊠

Consider the following production of an ag:

<expression>↓symtab ⟶ identifier↑tag declared↓symtab↓tag

It could be used to check that an identifier has been declared (i.e. tag $\in$ symtab). The use of identical attribute variables implies a copy of attribute values. For example, symtab appears in a defining attribute position of <expression> and in an applied attribute position of "declared". This indicates that the primitive predicate "declared" uses the <expression>'s symbol table.

Explaining the above definitions in terms of a derivation tree, we can see that values of synthesized attribute positions of a symbol X are defined in terms of attribute positions of the direct descendants of X; values of inherited attribute positions of X are defined in terms of attribute positions of its parent or siblings. The inherited attribute positions of the start symbol are given values in advance. Terminal symbols are not allowed to have inherited attribute positions since there is no subtree to which context can be transmitted. And, during parsing, the synthesized attribute values of terminal symbols are supplied by the scanner.

We now state two conditions that are necessary for an ag to be usable.

Definition 4.2.3 : An attributed grammar AG is well-formed if and only if

(1) every defining attribute position in a production is occupied by an attribute variable whose domain includes the domain of the attribute position and every applied attribute position is occupied by an attribute variable whose domain is a subset of the domain of the attribute position or by a constant attribute value in the domain of the attribute position.

(2) each attribute variable occuring in a production occurs in exactly one defining attribute position in that production. ⊠

Condition (1) guarantees that, during parsing, every attribute value is within the domain of the attribute position it occupies. Condition (2) ensures that, during parsing, every attribute position is assigned a value exactly once. Also note that, during parsing, once an attribute position is defined, its value is immediately available for use in applied positions.

The instance of a symbol X in a derivation together with its attribute values will be denoted by $X{\downarrow}i{\uparrow}s$ where i is an M_X-tuple of values, each value in the corresponding domain of i(X), and s is an N_X-tuple of values, each value in the corresponding domain of s(X). The notation $i \in i(X)$ means $i = (v_1, \ldots, v_{M_X})$, $i(X) = (d_1, \ldots, d_{M_X})$ and $v_k \in d_k$, $k = 1, \ldots, M_X$ ($s \in s(X)$ is similarly defined). We now consider the following sets:

$$AV_n = \{A{\downarrow}i{\uparrow}s \mid A \in \hat{V}_n,\ i \in i(A) \text{ and } s \in s(A)\}$$
$$AV_t = \{a{\uparrow}s \mid a \in \hat{V}_t \text{ and } s \in s(a)\}$$
$$AQ = \{q{\downarrow}i{\uparrow}s \mid q \in Q,\ i \in i(q) \text{ and } s \in s(q)\}$$
$$AV = AV_n \cup AV_t \cup AQ$$

Symbols in AV are termed <u>attributed symbols</u>. At times, we will consider a symbol together with inherited attribute values only. For this case, we define

$$AV_n^I = \{A{\downarrow}i \mid A \in \hat{V}_n \text{ and } i \in i(A)\}$$
$$AV_t^I = \hat{V}_t \text{ (since } M_a = \emptyset \text{ for all } a \in \hat{V}_t)$$
$$AQ^I = \{q{\downarrow}i \mid q \in Q \text{ and } i \in i(q)\}$$

$AV^I = AV_n^I \cup \hat{V}_t \cup AQ^I.$

We now formally define the concept of an <u>attributed derivation</u> in a well-formed attributed grammar AG.

<u>Definition 4.2.4</u> : Assume $\alpha, \beta \in AV^*$ and

$$A{\downarrow}a_1^\emptyset \ldots {\downarrow}a_{M_\emptyset}^\emptyset {\uparrow}b_1^\emptyset \ldots {\uparrow}b_{N_\emptyset}^\emptyset$$

$$\longrightarrow U_1{\downarrow}a_1^1 \ldots {\downarrow}a_{M_1}^1 {\uparrow}b_1^1 \ldots {\uparrow}b_{N_1}^1$$

$$\vdots$$

$$U_m{\downarrow}a_1^m \ldots {\downarrow}a_{M_m}^m {\uparrow}b_1^m \ldots {\uparrow}b_{N_m}^m$$

is a production in P. Then we have

$$\alpha A{\downarrow}i_\emptyset{\uparrow}s_\emptyset\gamma \Longrightarrow \alpha U_1{\downarrow}i_1{\uparrow}s_1 \ldots U_m{\downarrow}i_m{\uparrow}s_m\gamma$$

if and only if for $k = 1,\ldots,m$

(1) $i_\emptyset \in i(A)$.

(2) $s_k \in s(U_k)$.

(3) i_k is a value $(i_1^k,\ldots,i_{M_k}^k)$ such that (i) $i_j^k = a_j^k$ if a_j^k is a constant attribute value (ii) otherwise i_j^k is the value of the <u>unique</u> defining attribute position where a_j^k appears.

(4) $s_\emptyset$ is a value $(s_1^\emptyset,\ldots,s_{N_\emptyset}^\emptyset)$ such that (i) $s_j^\emptyset = b_j^\emptyset$

if $b_j^{\emptyset}$ is a constant attribute value (ii) otherwise $s_j^{\emptyset}$ is the value of the <u>unique</u> defining attribute position where $b_j^{\emptyset}$ appears.

(5) for any $U_k \in Q$ it is the case that

$$f_{U_k}(i_k) = (s_k, \underline{true}).$$

We also have $\alpha q{\downarrow}i{\uparrow}s\beta \Longrightarrow \alpha\beta$ for any $q{\downarrow}i{\uparrow}s \in AQ$ such that $f_q(i) = (s, \underline{true})$. ⊠

Informally, conditions (1) and (2) say that every defining attribute position has a value which is in its domain. Conditions (3) and (4) say that every applied attribute position has a value which is determined by the production that is used. Condition (5) says that the action functions associated to the primitive predicates on the right-hand side return the primitive predicates synthesized attribute values and the value <u>true</u> (i.e., they do not block the derivation).

As in the context-free case, we will use the notations $\overset{*}{\Longrightarrow}$, $\overset{+}{\Longrightarrow}$, $\underset{l}{\Longrightarrow}$ and $\underset{r}{\Longrightarrow}$. The language generated by a well-formed attributed grammar AG is

$$L(AG) = \{w \in AV_t^* \mid S{\downarrow}a{\uparrow}b \overset{*}{\Longrightarrow} w \quad \text{for some } a \in i(S) \text{ and } b \in s(S)\}$$

Example 4.2.1 : We now illustrate the above definitions using a small example. The following productions might be part of an attributed grammar AG_1 defining the evaluation of constant expressions

$$F\uparrow v_3 \longrightarrow F\uparrow v_1 \ / \ P\uparrow v_2 \ \text{div}\downarrow v_1 \downarrow v_2 \uparrow v_3 \quad (p1)$$
$$P\uparrow v_1 \longrightarrow \text{const}\uparrow v_1 \quad (p2)$$
$$F\uparrow v_1 \longrightarrow P\uparrow v_1 \quad (p3)$$

where the primitive predicate div has the following action function associated to it

```
div(v1, v2) : (integer, boolean)
      = if v2 = Ø
           then return(Ø, false)
           else return(v1/v2, true)
        fi
```

The following is an example of a leftmost derivation of $F\uparrow 3$ in AG_1. (The production that is applied to obtain a sentential form is indicated next to it.)

$$F\uparrow 3 \Longrightarrow F\uparrow 6 \ / \ P\uparrow 2 \ \text{div}\downarrow 6\downarrow 2\uparrow 3 \quad (p1)$$
$$\Longrightarrow P\uparrow 6 \ / \ P\uparrow 2 \ \text{div}\downarrow 6\downarrow 2\uparrow 3 \quad (p3)$$
$$\Longrightarrow \text{const}\uparrow 6 \ / \ P\uparrow 2 \ \text{div}\downarrow 6\downarrow 2\uparrow 3 \quad (p2)$$
$$\Longrightarrow \text{const}\uparrow 6 \ / \ \text{const}\uparrow 2 \ \text{div}\downarrow 6\downarrow 2\uparrow 3 \quad (p2)$$
$$\Longrightarrow \text{const}\uparrow 6 \ / \ \text{const}\uparrow 2$$

⌧

Since we are interested in guiding an error-correcting parser by context-sensitive information, we need schemes in which the evaluation of attributes can be interleaved with the parse. Such schemes are models for typical one-pass compilers. The class of ag we now consider is suitable for on-the-fly evaluation of attributes during a single left-to-right scan of the input string. This class is termed L-attributed grammars as defined by Lewis et al. [LRS 76] and has received considerable study.

Definition 4.2.5 : An attributed grammar AG is *L-attributed* if and only if

(1) it is well-formed.

(2) for each production $Y \longrightarrow Z_1 \ldots Z_m \in P$, it is the case that an attribute variable which appears in a synthesized attribute position of Z_j does not appear in any inherited attribute position of $Z_1, \ldots, Z_j$. ⊠

Condition (2) simply says that no attribute value is used to the left of the symbol which defines it.

A left-to-right parser for an attributed grammar AG is a context-free parser augmented by an *attribute stack* which is used for keeping track of attribute values as parsing progresses. The parser is constructed from the head grammar of AG.

Definition 4.2.6 : The head grammar H of AG is a cfg that is obtained as follows:

(1) the terminals of H are AG's terminals.

(2) the set of nonterminals of H includes AG's nonterminals and primitive predicates, and a set of copy symbols. Each copy symbol is associated with a sequence of operations on the attribute stack. Allowable operations are

top(t) which copies the top attribute stack element into t.

pop which pops the top attribute stack element.

push(t) which pushes the value of t onto the attribute stack.

(3) the set of productions P' of H is obtained from the productions of AG by removing attribute variables and constant attribute values. Copy symbols are added in the right-hand sides of these productions. Let I be the set of copy symbols. A production of the form $s \longrightarrow \epsilon$ is added to P' for each $s \in Q \cup I$.

(4) the start symbol of H is AG's start symbol. ⊠

Attribute stack manipulations are activated by the application of productions of the form $s \longrightarrow \epsilon$ where $s \in I \cup Q$. If $s \in Q$, the associated action function finds its input arguments on top of the attribute stack and returns its value on top of the attribute stack. If $s \in I$, the associated sequence of operations is executed. Also, the synthesized attribute values of an input symbol are pushed on the attribute stack as it is scanned.

Copy symbols are added in the right-hand sides in a way that guarantees that

(1) before a production is applied (in the LL case, this is before a right-hand side is predicted), the inherited attribute values of the left-hand side are on top of the attribute stack.

(2) after a production is applied (in the LL case, this is after a right-hand side has been recognized), the inherited and synthesized attribute values of the left-hand side are on top of the attribute stack. (That is, all other attribute values which were used during the application of the production are popped off.)

A head grammar with the above properties can be obtained automatically from AG. This construction is detailed in [Wat 77a; pp. 18-19]. The algorithm that is presented also tests if AG is L-attributed. We now illustrate the above definition.

__Example 4.2.2__ : Reconsider example 4.2.1 . It is easy to verify that AG_1, the fragment of grammar that was presented, is L-attributed. The head grammar of AG_1 is as follows:

F —> F / P div <#1>	(p1')
P —> const	(p2')
F —> P	(p3')
<#1> —> ε	(p4')
div —> ε	(p5')

where <#1> = (top(t); pop; pop; pop; push(t)) ⊠

The head grammar can then be used to construct an LL (or LR) parser. An attributed grammar is AF-LL(1) (respectively AF-LR(1)) if its head grammar is LL(1) (respectively LR(1)) [Wat 77a]. AF stands for attribute-free; this is because the parser is controlled entirely by the head grammar. The only syntactic role of the attributes is to signal context-sensitive errors via application of the action functions. The attributes never influence the flow of control of the parser other than making it detect a context-sensitive error. This technique is indeed very powerful. Watt was able to write an AF-LR(1) grammar specifying the complete syntax of PASCAL [Wat 77b].

However attributed grammars, as defined above, have a major disadvantage when they are to be used with locally least-cost correctors. It is often too late to do a correction by the time an action function evaluates to false. For example, we want to check that an identifier has been declared before doing a shift move which consumes it (so that we may delete the identifier or insert a string to its left). This is not possible with the scheme that was presented (because the identifier's synthesized attribute values are not available until after it is consumed). In the next section we consider how the above scheme can be modified slightly to allow earlier error detection.

4.3 Attribute-Free LL(1) Parsing

While we retain the separation of parsing and attribute evaluation by requiring the head grammar to be LL(1), we now allow inherited attribute values of the top stack symbol A and synthesized attribute values of the lookahead u to condition a prediction move of the AF-LL(1) parser. This is done by adding (by hand) shift-predicates to the grammar specification.

__Definition 4.3.1__ : Assume AG is an L-attributed grammar and production P_i is of the form

$$A{\downarrow}a_\emptyset{\uparrow}b_\emptyset \longrightarrow f{\uparrow}b_1\ U_2{\downarrow}a_2{\uparrow}b_2\ \ldots\ U_m{\downarrow}a_m{\uparrow}b_m$$

where $f \in \hat{V}_t$. Then a __shift-predicate__ for P_i is a total recursive function $s_i : i(A) \times s(f) \longrightarrow \{\underline{true}, \underline{false}\}$ and is used in the following way: P_i may be applied in a derivation only if $s_i(v_\emptyset, w_\emptyset) = \underline{true}$ where $v_\emptyset$ is the M_A-tuple of inherited attribute values of A and $w_\emptyset$ is the N_f-tuple of synthesized attribute values of f.

In the case the right-hand side of a production does not start with a terminal symbol (or an s-predicate does not specify a value for certain attribute values), a default value of __true__ is assumed. ⊠

The L-attributed restriction and the definition of the head grammar guarantee that the inherited attribute values of A can be found on top of the attribute stack when needed. The synthesized attributes of the lookahead are always available as they are supplied by the scanner. The class of AF-LL(1) grammars which are augmented by s-predicates is termed __SAF-LL(1)__. An SAF-LL(1) parser is presented in Appendix A.5 .

Example 4.3.1 : Consider the following SAF-LL(1) grammar AG_2, which will be used in all the examples that follow. This grammar defines a skeletal language in which identifiers may be "declared" and "used", in which no identifier may be declared more than once and in which no identifier may be used without being declared. Further, the set of allowable identifiers is {x, y}. For example, "dcl x dcl y use x end" is in L(AG_2), but "use y end" is not (x and y really are abbreviations for ident↑x and ident↑y).

Attribute Domains: ID = {x, y}; SYMTAB = 2^{ID}

Attribute Variables and their Domains

symtab, $symtab_1$, $symtab_2$, $symtab_3$: SYMTAB
id : ID

Terminals	Domains of Inherited Attribute Positions	Domains of Synthesized Attribute Positions
ident	—	ID
use	—	—
end	—	—
dcl	—	—
$	—	—
Non-Terminals		
<program>	—	—
<dec list>	SYMTAB	SYMTAB
<var dec>	SYMTAB	ID
<stmt list>	SYMTAB	—
<var>	SYMTAB	—
Primitive Predicate and Associated Action Function.		
declare	SYMTAB, ID	SYMTAB

declare(symtab : SYMTAB, id : ID) : (SYMTAB, boolean)

= return(symtab ∪ {id}, true)

Productions and s-predicates

(p1) <program> ⟶ $

<dec list>↓∅↑symtab

<stmt list>↓symtab

end

$

(p2) <dec list>↓symtab$_1$↑symtab$_3$

⟶ dcl

<var dec>↓symtab$_1$↑id

declare↓symtab$_1$↓id↑symtab$_2$

<dec list>↓symtab$_2$↑symtab$_3$

s_2(<dec list>↓symtab, dcl) = (symtab ≠ {x, y})

(p3) <dec list>↓symtab↑symtab

⟶ ϵ

(p4) <var dec>↓symtab↑id

⟶ ident↑id

s_4(<var dec>↓symtab, ident↑id) = (id ∉ symtab)

(p5) <stmt list>↓symtab ⟶ ϵ

(p6) <stmt list>↓symtab

⟶ use

<var>↓symtab

<stmt list>↓symtab

s_6(<stmt list>↓symtab, use) = (symtab ≠ ∅)

(p7) <var>↓symtab ⟶ ident↑id

s_7(<var>↓symtab, ident↑id) = (id ∈ symtab)

Note that s_2 prevents a declaration when the symbol table is full and s_6 prevents using a variable when the symbol table is empty.

Head Grammar (obtained automatically from the above for purposes of generating a parser)

(p1') <program> ⟶ $<#1> <dec list> <stmt list> end $ <#2>
(p2') <dec list> ⟶ dcl <var dec> <declare> <dec list> <#3>
(p3') <dec list> ⟶ <#4>
(p4') <var dec> ⟶ ident
(p5') <stmt list> ⟶ ϵ
(p6') <stmt list> ⟶ use <var> <stmt list>
(p7') <var> ⟶ id <#5>
(p8') <declare> ⟶ ϵ
(pk') <#k> ⟶ ϵ, k = 1,...,5

where <#1> = (push(∅))
<#2> = (pop; pop)
<#3> = (top(t); pop; pop; pop; push(t))
<#4> = (top(t); push(t))
<#5> = (pop)

The following is a parse of "$ use x end $" by the SAF-LL(1) parser corresponding to AG_2. A configuration of an SAF-LL(1) parser is a triple (σ, τ, w) where σ is the parse stack, τ is the attribute stack and w is the remaining input string.

```
1. σ = $ <#1> <dec list> <stmt list> end $ <#2>
   τ = ()
   w = $ use x end $

2. σ = <#1> <dec list> <stmt list> end $ <#2>
   τ = ()
   w = use x end $

3. σ = <dec list> <stmt list> end $ <#2>
   τ = (∅)
   w = use x end $

4. σ = <#4> <stmt list> end $ <#2>
   τ = (∅)
   w = use x end $

5. σ = <stmt list> end $ <#2>
   τ = (∅, ∅)
   w = use x end $
```

This is an *error configuration* since M(<stmt list>, *use*) = *predict* 6' and s_6(<stmt list>↓∅, *use*) = *false*. At this point the error corrector is invoked. This will be illustrated in the next section. ⊠

4.4 The Error Corrector

We are now ready to present a locally least-cost error corrector for a restricted but nevertheless interesting class of SAF-LL(1) parsers. We make the fundamental assumption that all attribute domains are finite. (In practice, any infinite attribute domains would be mapped into finite attribute classes.) The correction model we use is similar to the one of Definition 1.3.3, with the exception that symbols in terminal strings are now attributed. For this reason, the insertion and deletion costs are defined for attributed terminal symbols.

Error Correction Tables

We first consider the definition of (attributed) S and E tables. We now have $S : AV \longrightarrow AV_t^*$, where $S(A{\downarrow}i{\uparrow}s)$ is an optimal solution to

$$\min_{y \in AV_t^*} \{ IC(y) \mid A{\downarrow}i{\uparrow}s \stackrel{*}{\Longrightarrow} y \}$$

We also consider $E : (AV_n^I \cup \hat{V}_t) \times AV_t \longrightarrow AV_t^*$ where $E(A{\downarrow}i, a{\uparrow}s)$ is an optimal solution to

$$\min_{y \in AV_t^*} \{ IC(y) \mid A{\downarrow}i{\uparrow}s' \stackrel{*}{\Longrightarrow} y\ a{\uparrow}s \ldots \text{ and } s' \in s(A) \}$$

Note that the synthesized attribute values of A are not included in the domain of the E table since, as in the context-free case, an E value will constitute the final (i.e., rightmost) part of an inserted string. Algorithms for computing both the attributed S and E tables are given in Appendix A.6 .

Error Correction Procedure

As in the context-free case, we need a parser that has the IEDP. A technique similar to the one presented in Section 2.2 can be used. In this case, we want to restore both the parse stack and the attribute stack to the state they were in at the time the erroneous symbol was first encountered. This restoration can be done by buffering the transformations of both stacks in the auxiliary stack and later using a procedure "restore(σ, τ)" to undo parser moves.

A better way of obtaining the IEDP for the set of nonnullable SAF-LL(1) grammars has been developed by Fischer et al. [FTM 78]. An SAF-LL(1) grammar is nonnullable if and only if each production of its head grammar is of the form $A \longrightarrow X_1 \ldots X_k$ ($k \geq 1$) where $X_1 \ldots X_k \not\overset{+}{\Rightarrow} \epsilon$ or $A \longrightarrow \epsilon$. In this case, it is possible to check in advance if predicting an ϵ-production is correct. Since only predictions of ϵ-productions can possibly be erroneous, it is easy to construct an SAF-LL(1) parser which has the IEDP. Moreover, any SAF-LL(1) grammar can be algorithmically transformed into a nonnullable SAF-LL(1) grammar. This transformation is detailed in [FTM 78]. This second procedure has the advantage that linear time and space can be preserved even in the case the parse stack is not of bounded depth [FTM 78].

Assuming the IEDP is guaranteed, we first consider a function SAF-LL_Insert which computes a least-cost insertion string corresponding to error symbol $a\uparrow s \in AV_t$.

Before we exhibit the procedure let us introduce the notion of an <u>SAF-LL(1) error correction tree</u>. We process the parse stack $\sigma = X_1 \ldots X_p$ in a top-down fashion and as we consider $X_k \in \hat{V}_n \cup \hat{V}_t \cup Q$, we create the nodes at level k+1 in the tree. While processing X_k we have to consider its attribute values. The inherited attribute values of X_k will be at the top of a <u>local</u> attribute stack $N.\tau$ attached to node N of the tree. We assume $inh(X_k, N.\tau)$ is a function which returns these values. Also, we have to consider all possible synthesized attribute values of X_k which might be computed while expanding X_k, creating a node at level k+1 for each different choice (see Figure 4.4.2, lines 18-25 for $X_k \in \hat{V}_n \cup \hat{V}_t$ and lines 35-40 for $X_k \in Q$). Note that different least-cost expansions for X_k will be obtained for different inherited and synthesized attribute value combinations. The contents of a node N at level k+1 is a pair

(N.LC, N.τ) where

N.LC $\in AV_t^*$ is a least-cost insertion string which is derivable from $X_1 \ldots X_k$ and which satisfies all constraints imposed by primitive and shift predicates.

N.τ is a local attribute stack which takes into account the attribute stack manipulations which would take place assuming N.LC is inserted to the left of the error symbol. †

The error correction tree is built in such a way that all possible combinations of attribute values which might lead to a least-cost insertion are followed.

---

† A careful implementation of SAF-LL_Insert would not generate a separate local stack for each node. Rather, it would maintain a global tree structure of different attribute stack alternatives. N.τ being a pointer to a node in this tree.

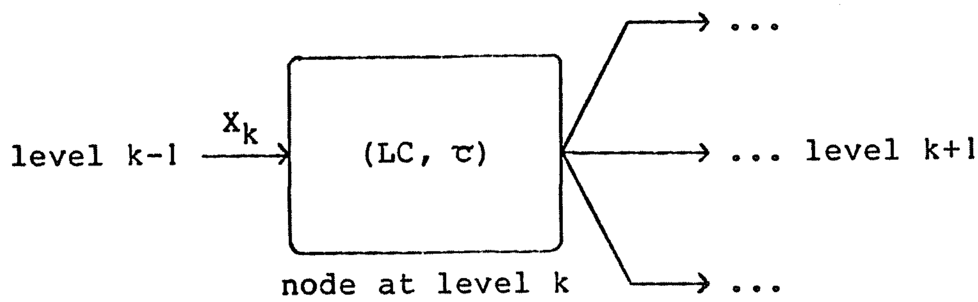


Figure 4.4.1 : the SAF-LL(1) error correction tree

The root of the error correction tree is (ε, τ) where τ is the restored attribute stack. The while loop in lines 5-44 creates the nodes in the tree until all nodes have an LC cost which is greater or equal to the cost of the lowest cost known insertion (Insert), or the bottom of the parse stack is reached. It also checks for possible insertions in lines 13-17.

```
function SAF-LL_Insert(σ, τ, a↑s) : AV_t* ;
   σ = X_1...X_p, the parse stack;
   (* X_1 is the top element in σ *)
   τ, the attribute stack;
   a↑s : AV_t, the error symbol;
type Node = record
              LC : AV_t*, least-cost insertion;
              τ, local attribute stack corresponding to LC;
```

```
        end Node;
var Level, NextLevel : set of Node;
    N, N' : Node;
    SV, EV , Insert : AV_t*
    τ' : attribute stack;
function inh(X : V̂_n u V̂_t u Q, τ : attribute stack) :
                                    M_X-tuple of attribute values;
   (* returns the inherited attribute values of X from τ,
   as outlined above *)

begin (* SAF-LL_Insert *)
  (* initialization *)
  Insert := ? ; k := 1 ;
  Level := {(ε, τ)}; NextLevel := ∅ ;
  (* main loop *)
  while ∃ N ∈ Level such that IC(N.LC) < IC(Insert)
        and k ≤ p do
     for all N ∈ Level such that IC(N.LC) < IC(Insert) do
        let N.τ = c_1...c_r;
     (* that is, the c_i's are stacked attribute values;
        c_r is the top element                         *)
        case X_k of
           Terminal, NonTerminal:
           (* check local correction *)
              EV := E(X_k↓inh(X_k, N.τ), a↑s)
              if IC(N.LC cat EV) < IC(Insert)
                 then Insert := N.LC cat EV
              fi;
           (* build next level in tree *)
              for all v ∈ s(X_k) do
                 SV := S(X_k↓inh(X_k, N.τ)↑v);
                 if IC(N.LC cat SV) < IC(Insert) then
                    NextLevel := NextLevel
```

```
                                  u {(N.LC cat SV, c_1...c_r v)}
              fi
           od;

        CopySymbol:
           let X_k = <#r>; τ' := <#r>(N.τ);
           (* i.e. τ' is obtained by applying <#r> *)
           if ∃ N' ∈ NextLevel such that N'.τ = τ'
             then if IC(N.LC) < IC(N'.LC)
                     then N'.LC := N.LC
                  fi
             else NextLevel := NextLevel u {(N.LC, τ')}
           fi;

        PrimitivePredicate:
           (v, pass) := f_{X_k}(inh(X_k, N.τ));
           if pass then
              NextLevel :=
                   NextLevel u {(N.LC, c_1...c_r v)}
           fi
        esac
     od;
     Level := NextLevel; NextLevel := ∅ ; k := k + 1
  end while;
  return(Insert)
end SAF-LL_Insert.
```

Figure 4.4.2 : function SAF-LL_Insert

Deletions can be implemented in the same way as in the LR(1) case (Chapter 3, Figure 3.1.1), with the difference that First_Occurence now ranges over AV_t. We therefore have:

```
procedure SAF-LL_Corrector(σ, τ, w);
   σ, the parse stack;
   τ, the attribute stack;
   w = a1...an$, the (preprocessed) remaining input string;
begin
   restore(σ, τ); (* if necessary *)
   y := ? ; i := 0;
   for all c ∈ AVt do
      if First_Occurence(c) ≠ absent then
         j := First_Occurence(c);
         z := SAF-LL_Insert(σ, τ, c);
         if D(j) + IC(z) < D(i) + IC(y)
            then y := z
                 i := j
         fi
      fi
   od;
   w := yai...an$;
   (* that is, delete a1...ai-1 and then insert y *)
end SAF-LL_Corrector.
```

Figure 4.4.3 : procedure SAF-LL_Corrector

Note that stack restoration (line 1) applies to both the parse stack and the attribute stack. Also note that, in practice, the same incremental approach that was presented in Section 3.2 would be used to advantage.

Example 4.4.1 : Reconsider grammar AG_2 given in Example 4.3.1 and assume all attributed terminal insertion costs are set to one. Further assume an SAF-LL(1) parser using SAF-LL_Corrector as an error corrector processes "$ use x end $". The parser detects an error when s_6 fails (step 5 in Example 4.3.1). After stack restoration, the error configuration is

σ = <#1> <dec list> <stmt list> end $ <#2>

τ = ()

w = use x end $

This configuration corresponds to step 2 in Example 4.3.1 . SAF-LL_Corrector first invokes SAF-LL_Insert(σ, τ, use). The error correction tree is given below (we indicate the parent of a node in parenthesis).

level 0: create the root of the tree

1. (ε, ())

Insert := ? (* initialization *)

level 1: X_1 = <#1>

2. (ε, (∅))

level 2: X_2 = <dec list>

3. (dcl x dcl y, (∅, {x,y})) (2)

   since S(<dec list>↓∅↑{x,y}) = dcl x dcl y

4. (dcl x, (∅, {x})) (2)

   since S(<dec list>↓∅↑{x}) = dcl x

5. (dcl y,(∅, {y})) (2)

   since S(<dec list>↓∅↑{y}) = dcl y

6. (ε, (∅, ∅)) (2)

   since S(<dec list>↓∅↑∅) = ε

level 3: X_3 = <stmt list>

Insert := dcl x dcl y

since E(<stmt list>↓{x,y}, use) = ε

and N_3.LC = dcl x dcl y

Insert := dcl x

since E(<stmt list>↓{x}, use) = ε

and N_4.LC = dcl x

7. (ε, (∅, ∅)) (6)

   since S(<stmt list>↓∅) = ε

level 4: X_4 = end

8. (end, (∅, ∅)) (7)

   since S(end) = end

At this stage a least-cost correction is obtained by insertion of "dcl x". SAF-LL_Corrector then considers deleting use and invokes SAF-LL_Insert(σ, τ, x). The reader may easily verify that the optimal insertion of "dcl" does not yield a lower cost correction. Since deleting "use x" costs 2, we know that an insertion of "dcl x" is optimal. We finally have corrected "$ use x end $" into "$ dcl x use x end $". ⊠

4.5 Properties of the Error Corrector

As mentioned in Section 1.3, a locally least-cost corrector can correct and parse any input string only if the parser has the correct prefix property. It is clear that an SAF-LL(1) parser has the correct prefix property if and only if any attributed symbol in AV that can be predicted can derive an attributed terminal string. The following definition and theorem present a procedure that decides if the correct prefix property holds.

Definition 4.5.1 : Let AG be an s-predicated attributed grammar with finite attribute domains. We define δ as a relation on AV_n such that $A{\downarrow}u{\uparrow}v \; \delta \; B{\downarrow}w{\uparrow}x$ if and only if $\alpha A{\downarrow}u{\uparrow}v\beta \Longrightarrow \alpha\gamma B{\downarrow}w{\uparrow}x\rho\beta$. ⊠

Note that the fact that we restrict ourselves to finite attribute domains guarantees the above definition is effective.

Theorem 4.5.1 : An SAF-LL(1) parser based on an SAF-LL(1) grammar AG with finite attribute domains has the correct prefix property if and only if for each $A{\downarrow}u{\uparrow}v \in AV_n$ such that $S{\downarrow}t{\uparrow}w\ \delta^*\ A{\downarrow}u{\uparrow}v$ for some $t \in i(S)$ and $w \in s(S)$ it is the case that $S(A{\downarrow}u{\uparrow}v) \neq ?$.

Proof : (If part) Assume that for any $A{\downarrow}u{\uparrow}v$ which can be predicted, we have $S(A{\downarrow}u{\uparrow}v) = y \neq$ "?". Then we have, by definition of S, $A{\downarrow}u{\uparrow}v \stackrel{*}{\Longrightarrow} y$ and therefore the correct prefix property can be guaranteed.

(Only if part) Assume $S{\downarrow}t{\uparrow}x\ \delta^*\ A{\downarrow}u{\uparrow}v$ and $S(A{\downarrow}u{\uparrow}v) = ?$ then we can be in a situation where $A{\downarrow}u{\uparrow}v$ is predicted and no attributed terminal string can be generated from it. Therefore the correct prefix property cannot be guaranteed. ⊠

Example 4.5.1 : Reconsider grammar AG_2 of Example 4.3.1 and assume the s-predicate s_6 is removed (s_6 prevents the prediction of use when the symbol table is empty). Then we can have the following attributed leftmost derivation:

<program> ⟹ $ <dec list>↓ø↑ø <stmt list>↓ø end $
⟹ $ <stmt list>↓ø end $
⟹ use <var>↓ø <stmt list>↓ø end $

Therefore we have <program> δ^* <var>↓ø. Further, we have S(<var>↓ø) = ? . So that AG_2 does not guarantee the correct prefix property if s_6 is omitted.

It should be noted that the presence or absense of s_6 does not change the language that is accepted by AG_2. It merely changes the point of error detection by an SAF-LL(1) parser. The above test can be helpful to a grammar designer to indicate when the specification of such s-predicates is needed. It is left to the reader to show that AG_2, as presented in Example 4.3.1, does in fact guarantee the correct prefix property. ⊠

We now prove the correctness of SAF-LL_Corrector and examine its efficiency in the general case and in the case of a bounded depth parse stack. The reader is invited to note the similarity between the following proofs and the corresponding proofs for the LL(1) case ([FMQ 77]) and the LR(1) case (Chapters 2 and 3).

Lemma 4.5.1 : Assume that during the execution of SAF-LL_Insert, a node N = (N.LC, N.τ) is added at level k of the error correction tree. Then it is the case that, if restarted in configuration (σ, τ, N.LC ...), the parser can accept N.LC giving an attribute stack of N.τ .

<u>Proof</u> : By induction on the number of levels that have been processed by SAF-LL_Insert.

Basis step: the Lemma trivially holds for (ϵ, τ), the root of the error correction tree.

Induction step: assume Lemma true at level k. Now consider a node N' at level k+1. N' can be added at level k+1 in one of three ways.

(1) If N' is added in lines 22-23 we have N' = (N.LC <u>cat</u> $S(X_k\downarrow inh(X_k, N.\tau)\uparrow v)$, $c_1 \ldots c_r v$) where N = (N.LC, $c_1 \ldots c_r$) is the parent node of N', at level k. By induction hypothesis, we know that N.LC can be accepted by the parser, giving an attribute stack of $c_1 \ldots c_r$. Now assume $S(X_k\downarrow inh(X_k, N.\tau)\uparrow v) = y$ (the condition in line 21 guarantees it is not "?"). There exists an attributed derivation of the form $X_k\downarrow inh(X_k, N.\tau)\uparrow v \overset{*}{\Longrightarrow} y$. Therefore N.LC <u>cat</u> y can be accepted by the parser, giving an attribute stack of $c_1 \ldots c_r v$, and the Lemma holds for N'.

(2) If N' is added in line 33, we have N' = (N.LC, <#r>(N.τ)) where N = (N.LC, N.τ) is the parent node of N', at level k and <#r> is the attribute stack transformation corresponding to copy symbol X_k. In this case, we merely simulate the transformation that would be done on the attribute stack by the parser. Therefore the Lemma holds for N' since it holds for N.

(3) If N' is added in lines 38-39, X_k is a primitive predicate and the proof that the Lemma holds for N' parallels the proof of case (1).

It follows immediately that the Lemma is true for all nodes in the error correction tree. ☒

<u>Lemma 4.5.2</u> : Assume that after reading and processing some input prefix $y $\in AV_t^*$ an SAF-LL(1) parser invokes SAF-LL_Insert with an error symbol of a↑s. During the execution of SAF-LL_Insert, wherever Insert contains a string z $\neq$?, it is the case that z a↑s can be accepted by the parser if it is restarted.

<u>Proof</u> : Aside from the initialization to "?" in line 2, Insert is assigned a value in only one place (line 16) and only when the new value has a cost less than the current value (and thus a cost < IC(?)). Insert is assigned a

value N.LC cat $E(X_k{\downarrow}inh(X_k, N.\tau), a{\uparrow}s)$ where $N = (N.LC, N.\tau)$ is a node in the error correction tree. By definition of E, we know $E(X_k{\downarrow}inh(X_k, N.\tau), a{\uparrow}s) = y$ is such that there exists an attributed derivation $X_k{\downarrow}inh(X_k, N.\tau){\uparrow}s' \overset{*}{\Longrightarrow} y\ a{\uparrow}s\ldots$. By Lemma 4.5.1, we know that N.LC can be accepted by the parser and will yield an attribute stack τ such that X_k's inherited attribute values are $inh(X_k, N.\tau)$. Therefore N.LC cat y a↑s can be accepted by the parser. ⊠

Theorem 4.5.2 : Consider some SAF-LL(1) grammar AG with finite attribute domains. Assume that, after reading and processing some input prefix $\$x \in AV_t^*$, the corresponding SAF-LL(1) parser invokes SAF-LL_Insert with error symbol a↑s as soon as a↑s is encountered. Then SAF-LL_Insert will return a string $y \in AV_t^+ \cup \{?\}$ such that y is an optimal solution to

$$\min_{y \in AV_t^+ \cup \{?\}} \{\ IC(y) \mid (\$xya{\uparrow}s \text{ can be accepted by the parser}) \text{ or } (y = ?)\ \}$$

Proof : By Lemma 4.5.2, we know any string ≠ ? assigned to Insert is correct and a new value is assigned to Insert only if it is of a lower cost than the current value. We need only therefore show that at some point an attempt to assign a string of cost IC(y) must be made. If y = ?, Insert is assigned value "?" (line 2) and will never be assigned a different value since y is least-cost. Otherwise we will show how SAF-LL_Insert traces the various ways y a↑s may be recognized once parsing is restarted.

Assume $y = y_1 y_2$. The induction hypothesis here is that if y_1 is generated from $X_1 \ldots X_k$ and if processing has not halted yet then at level k we have a node N = (LC, τ) such that IC(LC) = IC(y_1) and τ would be the attribute stack after y_1 was recognized from $X_1 \ldots X_k$.

Initial step: write y a↑s as $y_1 y_2$ a↑s and assume $y_1 = \epsilon$. Then it is the case that $N_1 = (\epsilon, \tau)$, the sole node at level 1, is such that IC(y_1) = IC(N_1.LC) and that an attribute stack of τ is obtained if y_1 is inserted and later parsed.

Iterative step: assume y a↑s is written as $y_1 y_2$ a↑s and we have just completed processing X_{k-1} creating nodes at level k. By induction hypothesis we know that there exists a node N = (N.LC, N.τ) at level k such that IC(y_1) = IC(N.LC) and y_1 can be generated from some $X_1{\downarrow}a_1{\uparrow}b_1 \ldots X_{k-1}{\downarrow}a_{k-1}{\uparrow}b_{k-1}$ producing an attribute stack of N.τ. We continue by tracing how y_2 a↑s might be recognized. It may be the case that y_2 a↑s is fully generated by $X_k{\downarrow}inh(X_k,\ N.\tau){\uparrow}v$ where $v \in s(X_k)$ and N is a node at level k. Then it must be that IC(y_2) = IC(E($X_k{\downarrow}inh(X_k,\ N.\tau){\uparrow}v$, a$\uparrow$s)), else y is not least-cost. In this case Insert is assigned a string of cost $IC(y_1)+IC(y_2) = IC(y)$ since y_2 must be least cost (line 16). Otherwise write y_2 a$\uparrow$s as $z_1 z_2$ a$\uparrow$s and assume $z_1 \in AV_t^*$ is generated from $X_k \downarrow inh(X_k, N.\tau) \uparrow v$ where N is a node at level k and $v \in s(X_k)$. We now consider three different cases:

(1) If X_k = <#r>, a copy symbol, then we have $z_1 = \epsilon$. In this case we create a node N' = (N.LC, <#r>(N.τ)) at level k+1 (lines 26-34).

(2) If $X_k \in Q$, it must be the case that $f_{X_k}(inh(X_k, N.\tau))$ = (v, <u>true</u>) since, by Lemma 4.5.1, N.LC is correct and we create a node N' = (N.LC, N.τ v) at level k+1 (lines 38-39).

(3) If $X_k \in \hat{V}_t \cup \hat{V}_n$, we create a node N' = (N.LC <u>cat</u> $S(X_k \downarrow inh(X_k, N.\tau) \uparrow v)$, N.$\tau$ v) (lines 22-23) where $IC(S(X_k \downarrow inh(X_k, N.\tau) \uparrow v)) = IC(z_1)$ since z_1 is assumed least-cost.

In all cases, we created a node N' = (LC', τ') at level k+1 such that $IC(LC') = IC(y_1 z_1)$ and τ' is the attribute stack which would be obtained by processing $y_1 z_1$. If IC(Insert) > IC(LC') this step is repeated for level k+1 with $y_1 z_1$ renamed y_1 and z_2 a$\uparrow$s renamed y_2 a$\uparrow$s. If IC(Insert) $\leq$ IC(LC') the algorithm may terminate but a least-cost Insert must already have been found since IC(LC')

$\leq$ IC(y).

The iterative step is repeated until the symbol which finishes the recognition of y a↑s is processed or until IC(Insert) is less or equal to the cost of all LC values. In either case a simple induction on the number of iterative steps executed establishes that an Insert value of cost IC(y) must be obtained. ⊠

<u>Theorem</u> <u>4.5.3</u> : Consider some SAF-LL(1) grammar AG with finite attribute domains and such that the correct prefix property can be guaranteed. Assume that for some input string $z = \$xa_1 \ldots a_n\$ \in AV_t^*$, $\$x\ldots \in L(AG)$ but $\$xa_1 \ldots \notin L(AG)$. Further assume that while attempting to parse z an SAF-LL(1) parser invokes SAF-LL_Corrector as soon as a_1 is encountered. Then SAF-LL_Corrector will delete $a_1 \ldots a_i$ and insert y such that (i, y) is a solution to

$$\min_{\emptyset \leq i' \leq n} \{ \min_{y' \in AV_t^*} \{ DC(a_1 \ldots a_{i'}) + IC(y') \mid \$xy'a_{i'+1} \ldots \in L(AG) \} \}$$

<u>Proof</u> : similar to proof of Theorem 3.2.1 . ⊠

Before we examine the complexity of SAF-LL_Corrector, we state a result concerning the complexity of SAF-LL_parser.

Theorem 4.5.4 : Given an SAF-LL(1) grammar, the corresponding SAF-LL(1) parser requires O(n) time to parse a correct input string of length n if we assume each evaluation of a s-predicate or action function takes no more than a constant time.

Proof : Follows directly from the linerarity of LL(1) parsers [AU 73; Volume 1] and the fact that the modifications only block (but do not otherwise change) the actions of a normal LL(1) parser. † ⊠

Lemma 4.5.3 : The number of nodes created in a given invocation of SAF-LL_Insert at any level k in the error correction tree is bounded by a constant depending solely on the grammar.

Proof : Let $\sigma = X_1 \ldots X_k \ldots X_p$ be the parse stack and C_1 be the number of distinct tuples of synthesized attribute values over any s(X) for $X \in \hat{V}$. We know that C_1 is finite

---

† If the evaluation of s-predicates and action functions cannot be done in constant time, we can still guarantee that the number of parser moves is O(n).

since we assume finite attribute domains. Now consider two cases.

(1) First assume X_k terminates the recognition of the right-hand side of some production $P_i = (Y \longrightarrow Z_1 \ldots Z_m)$ (that is $Z_m = X_k \downarrow a_k \uparrow b_k$). All nodes at level k+1 have a local attribute stack configuration of the form (..., inh(Y), syn(Y)) where inh(Y) is a tuple of inherited attribute values of Y and syn(Y) is a tuple of synthesized attribute values of Y. We note that, by construction of the head grammar (Section 4.2), (..., inh(Y)) is the unique sequence of attribute values that was already present on the attribute stack σ at the time SAF-LL_Insert was invoked. (..., inh(Y)) was fixed at the time P_i was predicted, before the error was detected. Therefore the number of nodes at level k of the tree is bounded by C_1 since local attribute stacks can only differ in their syn(Y) part and there cannot be two nodes with identical attribute stacks at the same level in the tree.

(2) Now consider a level k' which does not correspond to the above category (i.e. such that $X_{k'}$ does not complete the recognition of a right-hand side). We can clearly have no more than maxrhs symbols of this class before a stack symbol which terminates a right-hand side is encountered. Now copy symbols and primitive predicates do not increase

the number of nodes at the next level. Grammar symbols can add at most C_1 new nodes for a given node (i.e., each possible tuple of synthesized attribute values for a given tuple of inherited attribute values). Thus each level can increase by at most a factor C_1 nodes and therefore we have at most $C_1 . C_1^{maxrhs}$ distinct nodes before a stack symbol of class (1) is encountered. ⊠

<u>Lemma</u> <u>4</u>.<u>5</u>.<u>4</u> : Assume an SAF-LL(1) parser using SAF-LL_Corrector as an error corrector processes \$x\$ and corrects it into \$x'\$. Then it is the case that $|x'| = O(|x|)$.

<u>Proof</u> : We need only show that each symbol inserted during error correction can be charged to some input symbol and that each input symbol is charged for at most a constant number of insertions.

For charging purposes we associate each parse stack symbol with the input symbol which caused it to be pushed on the parse stack. It is easy to show that, during normal parsing, the number of stack symbols so charged to a given input symbol is bounded by a constant.

Now assume SAF-LL_Corrector is invoked with error symbol a↑s and a parse stack of $\sigma = X_1 \ldots X_p$. Consider the particular invocation of SAF-LL_Insert which yields the optimal insertion. Starting with X_1, stack symbols are examined and each stack symbol either generates a least-cost string or a least-cost prefix string. In either case the length of the insertion string associated with a given symbol can be bounded by a grammar-dependent constant. Now considering the fact that, when parsing is resumed, those stack entries which generate least-cost strings are effectively deleted, we can charge these portions of LC strings to their corresponding stack symbols. Further, that stack symbol $X_j{\downarrow}a_j{\uparrow}b_j$ which derives $a{\uparrow}s$ is in effect replaced by $w \in AV^*$ where $X_j{\downarrow}a_j{\uparrow}b_j \overset{*}{\underset{I}{\Longrightarrow}} p\ A{\downarrow}i_A{\uparrow}s_A\ q \underset{I}{\Longrightarrow} px\ a{\uparrow}s\ yq$, $w = yq$ and px is the least-cost prefix to be inserted. Since w is determined solely by $X_j{\downarrow}a_j$ and $a{\uparrow}s$, its size can be bounded by a grammar-dependent constant and we can associate these stack symbols to $a{\uparrow}s$. ⊠

<u>Lemma 4.5.5</u> : Assume a bounded depth parse stack SAF-LL(1) parser using SAF-LL_Corrector processes \$x\$. Then stack restoration requires at most $O(|x|)$ time and space in all.

<u>Proof</u> : Let $\alpha \in V^*$ be the stack symbols <u>just before</u> buffering begins. As in normal LL(1) parsing, the number of moves induced by an error symbol and a given parse stack symbol is

bounded by a constant. Thus since $|\alpha|$ is bounded by a constant, so is the total number of moves buffered in the auxiliary stack AS. Since attribute stack and parse stack manipulations can be undone in constant time, procedure restore requires only a constant time per invocation and at most $O(|x|)$ time in all. The $O(|x|)$ space bound is trivial. ☒

<u>Theorem 4.5.5</u> : Assume we are given an SAF-LL(1) grammar AG that satisfies the following conditions:

(1) all attribute domains are finite.

(2) the correct prefix property can be guaranteed.

(3) each evaluation of a s-predicate or action function takes no more than a constant time.

Then processing the input x with the corresponding SAF-LL(1) parser and SAF-LL_Corrector requires

(1) at most $O(|x|^2)$ time and $O(|x|)$ space in the general case.

(2) at most $O(|x|)$ time and space if a bounded depth parse stack is assumed.

<u>Proof</u> : (1) In the general case, for each $|x|$ possible errors it may take $O(|x|)$ time and space to restore both the parse stack and the attribute stack (the same argument used in Theorem 2.2.1 applies to both stacks). We now show that every invocation of SAF-LL_Insert takes $O(|x|)$ time. Con-

sider the while-loop in lines 5-44. The number of times it is executed is bounded by $O(|\sigma|) = O(|x|)$ and, given a careful implementation of local attribute stacks and LC strings, each execution takes at most constant time. This is because any node at a given level can be processed in no more than a constant time and, by Lemma 4.5.3, we know that there are at most a constant number of nodes at a given level. Since SAF-LL_Insert is invoked for each $a \in AV_t$, at most, it follows immediately that SAF-LL_Corrector takes time $O(|x|)$ for each correction and therefore time $O(|x|^2)$ in all. The $O(|x|)$ space bound is trivial and the desired result follows immediately.

(2) In the case of a bounded depth parse stack, one invocation of SAF-LL_Insert can process the entire (bounded depth) parse stack in constant time, using an amount of space bounded by a constant. Therefore one invocation of SAF-LL_Corrector requires constant time and space. Moreover stack restoration takes $O(|x|)$ additional time and space in all (Lemma 4.5.5). ⊠

Chapter 5 : CONCLUSIONS

5.1 Summary

A goal of this research was to extend the FMQ LL(1) error corrector [FMQ 77 and FM 77] to be usable with a large class of practical parsers (viz, the LR(1)-based class). Although the problem of error correction has previously received much attention, most of the other techniques suffer very serious drawbacks. Very often, they fail when faced with certain syntax errors and are forced to skip ahead in the input stream, completely ignoring portions of it. Further, in most of the cited work, the issue of time and space complexity is ignored. Indeed many published techniques exhibit non-linear behavior.

The work presented in this thesis has both theoretical and practical significance. The error correction model introduced by Fischer et al. [FMQ 77 and FM 77] has been presented and extended to the LR(1) and SAF-LL(1) parsers. For all of these techniques, a locally least-cost correction is guaranteed and, in cases of special interest (e.g. bounded depth parse stack), linearity can be established.

On the practical side, preliminary experience with the LL(1) and LR(1) correctors indicate that these can be used to advantage with most LL- or LR-driven compilers. Both correctors can operate satisfactorily with a rather small primary storage requirement. Although the SAF-LL(1) corrector has not yet been implemented, there is good reason to believe that context-sensitive information can help in providing the user with highly plausible corrections.

All of the techniques developed here have the fundamental advantage that the introduction of error correction in the translation process has very little impact on the overall structure of a compiler. This is a direct consequence of the locality of our correction model.

5.2 Directions for Future Research

This research presents a structured approach to error correction for a number of practical parsers. It seems very likely that locally least-cost correctors can be developed for other classes of parsers.

The generalized left-corner (GLC) parsing technique described by Demers [Dem 77] subsumes the LL and LR techniques. By combining the FMQ algorithm and the LR corrector of Chapter 2, one can hope to develop an error corrector for GLC parsers, which would certainly require smaller tables then the LR corrector.

The techniques of Chapters 2 and 4 could be combined to generate an error corrector for AF-LR(1) parsers.

Attributed error correction in the presence of infinite attribute domains needs to be fully investigated. It is our feeling that infinite (or large) domains are used in a rather restricted manner in the definition of common programming languages (e.g. for keeping track of identifiers in a symbol table and for counting the number of elements in a linear list). An approach that may prove fruitful is to delay some of the computations of the S and E values until parse time.

In a recent Ph.D. thesis, Poplawski [Pop 78] has extended the FMQ LL(1) corrector to the LL-regular parsing technique which uses a regular lookahead to make parsing decisions. Parsing is done in two passes: in a first pass the input program is processed in reverse by a generalized sequential machine, and in a second pass the modified text is processed by a top-down parser. This allows the introduction of non-local information via the regular lookahead into a locally least-cost error correction scheme. A syntax error such as a missing "if" (Example 3.3.1) can then be reported and corrected without backing up. It seems likely that the LR(1) corrector of Chapter 2 can be extended to work with LR-regular parsers.

Finally, it is our belief that syntactic error correction applies to more than just compilers. For example, tools for high-level programming might include a specialized text editor that understands the syntax of the programming language on which it is based. For such a text editor, good diagnostic and correction capabilities are of much interest. In this case costing could be used as a basis for providing a list of plausible corrections to the user.

APPENDIX

A.1 S and E Tables Calculation

Given an augmented cfg G, the following procedures compute the S and E tables as defined in Section 1.4 . Correctness and efficiency of STable and ETable are discussed in [FMQ 77].

```
procedure STable;
begin
    (* initialization *)
      for all a ∈ V̂_t do S(a) := a od;
      for all A ∈ V̂_n do S(A) := ? od;
    (* main loop *)
      repeat
         NoChange := true;
         for all (A ⟶ X_1...X_n) ∈ P do
            if IC(X_1...X_n) < IC(A)
               then S(A) := S(X_1...X_n)
                    NoChange := false;
            fi
         od
      until NoChange
end STable.
```

```
procedure ETable;
begin
   (* initialization *)
     for all A ∈ V̂ do
        for all a ∈ V̂_t do E(A,a) = ? od
     od;
     for all a ∈ V̂_t do E(a,a) := ∈ od;
   (* main loop *)
     repeat
        NoChange := true;
        for all a ∈ V̂_t do
           for all (A ⟶ X_1...X_n) ∈ P do
              cost := min  (IC(X_1...X_{i-1})+IC(E(X_i,a))) ;
                      1 ≤ i ≤ n
             (* j is the value giving the above min *)
              if cost < IC(E(A, a))
                 then E(A,a) := S(X_1...X_{j-1}) cat E(X_j,a);
                      NoChange := false
              fi
           od
        od
     until NoChange
end ETable.
```

A.2 CFSM Construction Algorithm

Given an augmented cfg G, CFSM constructs the characteristic finite state machine [DeR 71].

```
procedure CFSM(G);
begin
  s∅ := {[S' ⟶ $•S$]}; marked[s∅] := false;
  while ∃ s ∈ 𝐒 such that not marked[s] do
     let s ∈ 𝐒 such that not marked[s];
     marked[s] := true;
    (* compute closure of s *)
     for all I = [A ⟶ α•Bγ] ∈ s do
        for all B ⟶ δ ∈ P' such that [B ⟶ •δ] ∉ s do
           s := s ∪ [B ⟶ •δ]
        od
     od;
    (* compute transitions out of s *)
     for all X ∈ V̂ do
        T := {[A ⟶ αX•γ] | [A ⟶ α•Xγ] ∈ s}
        if T ≠ ∅ and ( ∀ s' ∈ 𝐒, T ≠ basis(s'))
           then basis(s") := T ;
                marked[s"] := false ;
                GOTO(s, X) := s" ;
        fi
     od
  end while
end CFSM.
```

A.3 Bottom Up Stack Traversal LR Error Corrector

The following function computes an insertion string having the same properties as the one computed by LR_Insert in Section 2.4 . However it computes the insertion from right to left while examining the parse stack in a bottom up fashion.

```
function BU_LR_Insert(σ, a) : TerminalString;
   σ = s_0...s_p, the parse stack;
   a ∈ V̂_t, the error symbol;
begin
  CURSTAGE := STAGE(s_0);
  for all i such that I_i ∈ basis(s_0) do
     CURSTAGE.IS_i := ?
  od;
  for k := 1 to p do
     PREDSTAGE := CURSTAGE; CURSTAGE := STAGE(s_k);
     for all n such that I_n ∈ basis(s_k) do
      (* link I_n to predecessors in basis(s_k-1) *)
        CURSTAGE.IS_n := ? ;
        let m be such that (m, s_k-1) ∈ Pred(I_n);
        if I_m ∈ closure(s_k-1)
           then (*follow back-ptrs to basis items *)
              for all (b(i),y_i) ∈ B(I_m) do
                 if IC(y_i cat PREDSTAGE.IS_b(i))
                    < IC(CURSTAGE.IS_n)
                    then CURSTAGE.IS_n
                           := y_i cat PREDSTAGE.IS_b(i);
                 fi
```

```
                od;
              (* check for local insertion *)
                if IC(T(Im, a)) < IC(CURSTAGE.ISn) then
                   CURSTAGE.ISn := T(Im, a)
                fi
             else (* we have Im ∈ basis(sk-1) *)
                if IC(CURSTAGE.ISn) > IC(PREDSTAGE.ISm)
                   then CURSTAGE.ISn := PREDSTAGE.ISm
                fi
          fi
       od
    od;
    INSERT := ?  ;
    for all i such that [Ai ⟶ αi•βi] ∈ basis(sp) do
       if IC(INSERT) > IC(Insert(βi, a)) then
          INSERT := Insert(βi, a)
       fi;
       if IC(INSERT) > IC(S(βi) cat ISi) then
          INSERT := S(βi) cat ISi
       fi
    od;
    return( INSERT )
end BU_LR_Insert.
```

Correctness of this function can be obtained in the same way as that of LR_Insert. Simply notice that IS_i in the stage corresponding to stack state s_j is a least-cost terminal string that can be used to the left of error symbol "a" if item $I_i \in basis(s_j)$ is to be used during the parse of the string to be inserted. Also note that this function uses the same tables as LR_Insert.

A.4 PASCAL IC and DC Functions

The following insertion and deletion costs were used for testing LR_Corrector using PASCAL programs.

Terminal	IC	DC
$	500	—
program	1	1
ID	10	20
(	8	20
downto	4	20
)	7	20
;	2	20
to	4	20
.	10	20
,	2	15
for	15	25
not	7	20
nil	20	20
MULTOP	5	20
or	5	20
RELOP	5	20
label	10	20
then	4	25
if	15	20
CONSTANT	9	20
const	10	20
else	10	20
with	10	20
=	3	20
while	9	20
do	4	20
STRING	12	20

Terminal	IC	DC
+	3	20
-	3	20
until	8	20
repeat	10	20
CHARACTER	12	20
type	18	20
goto	6	20
:=	6	20
begin	8	20
function	12	20
procedure	10	20
forward	20	20
var	10	20
↑	8	20
case	9	20
:	2	20
file	15	20
..	3	20
set	15	20
end	6	20
packed	15	20
array	10	20
[	7	20
record	10	20
]	6	20
of	1	20

A.5 The SAF-LL(1) Parser

The following procedure is an SAF-LL(1) parser as described in Section 4.3 .

```
procedure SAF-LL_Parser(options † : i(S')) ;
var σ = X1...Xp, the parse stack;
    τ, the attribute stack;
    synX1 : s(X1);
  (* M is the LL(1) parsing table of H, the head grammar *)
    M : array[1..|V̂n u Q u I|, 1..|V̂t|]
                                     of Prediction u {error}

begin (* SAF-LL_Parser *)
    σ := S'; τ := (options);
    repeat
       let a↑s be the attributed lookahead;
       case X1 of
          NonTerminal:
             if M(X1, a) = predict j
                and sj(X1↓inh(X1, τ), a↑s)
                   then σ.pop; σ.push(RHSj)
                   else SAF-LL_Corrector
             fi;
```

† The inherited attributes of the start symbol are usually equivalent to options in a typical compiler.

```
        Terminal:
            if X1 = a
                then σ.pop; τ.push(s);
                     shift to next input symbol;
                else SAF-LL_Corrector
            fi;

        CopySymbol:
            let X1 = <#r>;
            τ := <#r>(τ);

        PrimitivePredicate:
            (synX1, pass) := f_X1(inh(X1, τ));
            if pass
                then τ.push(synX1)
                else SAF-LL_Corrector
            fi
    esac
  until σ = ∅
end SAF-LL_Parser.
```

A.6 Attributed S and E Tables Calculations

The following procedure STable computes the attributed S table as defined in Section 4.4 . The main procedure is similar to the STable procedure given in Appendix A.1 for the context-free case. ReEvalS(P_i, s_i) considers the reevaluation of S($LHS_i \downarrow u \uparrow v$) for all u $\in$ i(LHS_i) and v $\in$ s(LHS_i), using production i. SearchProdS is a recursive procedure whih assigns values to attribute positions of the symbols in RHS_i by doing a depth-first search of a tree which can be built by considering all possible combinations of attribute values.

The attribute values of a prefix of a production P_i are kept in the arrays v_k and w_k, which are used as stacks in the depth-first search.

```
procedure STable(AG);
   AG, an  SAF-LL(1) grammar;
function ReEvalS; (* see next page *)

begin (* STable *)
   (* initialization *)
   for all A↓i↑s ∈ AV_n do S(A↓i↑s) := ? od;
   for all a↑s ∈ AV_t do S(a↑s) := a↑s od;
   for all q↓i↑s ∈ AQ do
      S(q↓i↑s) := if f_q(i) = (s, true) then ∈ else ? fi
   od;

   (* main loop *)
   repeat
      NoChange := true;
      for all (P_j, s_j) ∈ P do
         NoChange := NoChange and not ReEvalS(P_j,s_j)
      od
   until Nochange
end STable.
```

function ReEvalS(P_i, s_i) : boolean ;

$P_i = (A{\downarrow}a_\emptyset{\uparrow}b_\emptyset \longrightarrow B_1{\downarrow}a_1{\uparrow}b_1 \ \ldots\ B_m{\downarrow}a_m{\uparrow}b_m)$;

s_i = s-predicate;

(* assume $a_k = (a_1^k,\ldots,a_{M_k}^k)$

$b_k = (b_1^k,\ldots b_{N_k}^k)$ for $k = \emptyset,\ldots,m$ *)

var Change : boolean;

v_k : domains(a_k), $k = \emptyset,\ldots,m$;

w_k : domains(b_k), $k = \emptyset,\ldots,m$;

(* where domains(a_k) is a tuple of domains defined as follows. If a_j^k is an attribute variable then the j's component of domains(a_k) is i(a_j^k), otherwise it is $\{a_j^k\}$; domains(b_j) is defined in a similar manner (i.e., it is either s(b_j^k) or $\{b_j^k\}$) *)

procedure SearchProdS; (* see next page *)

begin (* ReEvalS *)

Change := **false**;

for all $v_\emptyset \in$ domains($a_\emptyset$) **do**

    if $m = \emptyset$ **then** (* ϵ-production *)

        copy $w_\emptyset$ from defining position;

        $S(A{\downarrow}v_\emptyset{\uparrow}w_\emptyset) := \epsilon$;

        Change := **true**

    else copy v_1 from defining position;

          SearchProdS(1, ϵ)

    fi

od;

return(Change)

end ReEvalS.

```
procedure SearchProdS(j, LC);
   j : 1..m, level in the tree;
   LC : AV*_t, least-cost string derivable from B_1...B_{j-1};
   T : AV*_t;
begin
  for all w_j ∈ domains(b_j) do
     if j = 1 and B_1 ∈ V̂_t then
     (* check s-predicate *)
        if not s_i(A↓v_∅, B_1↑w_1) then goto continue fi
     fi;
     T := S(B_j↓v_j↑w_j);
     if T ≠ ? then
        LC := LC cat T;
        if j < m then copy v_{j+1} from defining position;
                      SearchProdS(j+1, LC)
        else copy w_∅ from defining position;
             if IC(LC) < IC(S(A↓v_∅↑w_∅))
                then S(A↓v_∅↑w_∅) := LC ;
                     Change := true
             fi
        fi;
     continue : od
end SearchprodS.
```

The following procedure ETable computes the attributed E table as defined in Section 4.4 . The main procedure is similar to the ETable procedure given in Appendix A.1 for the context-free case. ReEvalE(P_i, s_i, a↑s) considers the reevaluation of E(LHS_i↓u, a↑s) for all u ∈ i(LHS_i). SearchProdE is a recursive procedure which assigns attribute values in the same way as SearchProdS.

```
procedure ETable(AG);
   AG, an  SAF-LL(1) grammar;
function ReEvalE; (* see next page *)

begin
     (* initialization *)
     for all A↓i ∈ AV_n^I,
            all a↑s ∈ AV_t do
        E(A↓i, a↑s) := ?
     od;
     for all a↑s ∈ AV_t do
        E(a, a↑s) := ∈
     od;

     (* main loop *)
     repeat
        NoChange := true;
        for all a↑s ∈ AV_t,
               all (P_j, s_j) ∈ P do
          NoChange := NoChange and not ReEvalE(P_j, s_j, a↑s)
        od
     until Nochange
end ETable.
```

```
function ReEvalE(Pi, si, a↑s) : boolean;
   Pi = (A↓a∅↑b∅ —> B1↓a1↑b1 ... Bm↓am↑bm)
   si = s-predicate;
   a↑s : AVt, the error symbol;
var Change : boolean;
    vk : domains(ak), k = ∅,...,m;
    wk : domains(bk), k = ∅,...,m;
    (* as defined in ReEvalS *)
procedure SearchProdE; (* see next page *)

begin (* ReEvalE *)
   Change := false;
   if m > ∅ then
      for all v∅ ∈ domains(a∅) do
         copy v1 from defining position;
         SearchProdE(1, ∈)
      od
   fi;
   return(Change)
end ReEvalE.
```

```
procedure SearchProdE(j, LC);
   j : 1..m, level in the tree;
   LC : AV*_t, least-cost string derivable from B_1...B_{j-1};
   T : AV*_t;
begin
   (* check for a local correction *)
   T := LC cat E(B_j↓v_j, a↑s);
   if IC(T) < IC(E(A↓v_Ø, a↑s)) then
      if j ≠ 1 or B_1 ∉ V̂_t or s_i(A_Ø↓v_Ø, AFirst(T))
         (* where AFirst(T) returns the first
            attributed symbol in T *)
         then E(A↓v_Ø, a↑s) := T;
              Change := true
      fi
   fi;

   (* recursively invoke procedure at next level *)
   if j < m then
      for all w_j ∈ domains(b_j) do
         if j ≠ 1 or B_1 ∉ V̂_t or s_i(A_Ø↓v_Ø, B_1↑w_1)
            then T := LC cat S(B_j↓v_j↑w_j)
               if IC(T) < IC(E(A↓v_Ø, a↑s))
                  then copy v_{j+1} from defining position;
                       SearchProdE(j+1, T)
               fi
         fi
      od
   fi
end SearchProdE.
```

BIBLIOGRAPHY

[AP 72] Aho, A. V. and T.G. Peterson, "A Minimum Distance Error-Correcting Parser for Context-Free Languages," SIAM Journal on Computing, Vol. 1, No. 4, pp. 305-312, December 1972.

[AHU 76] Aho, A. V., J. E. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, Reading, MA, 1976.

[AU 73] Aho, A. V. and J. D. Ullman, The Theory of Parsing, Translation and Compiling, Volume 1 : Parsing, Volume 2 : Compiling, Prentice Hall, Englewood Cliffs, NJ, 1973.

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

[BAC 77] Backhouse, R. C., "String to String Correction and Correction of Regular Languages," Technical Report, Department of Computer Science, Heriot-Watt University, Edinburgh, November 1977.

[CM 63] Conway, R. W. and W. L. Maxwell, "CORC, the Cornell Computing Language," Communications of the ACM, Vol. 6, No. 6, pp. 317-324, July 1963.

[CW 73] Conway, R. W. and T. R. Wilcox, "Design and Implementation of a Diagnostic Compiler for PL/I," Communications of the ACM, Vol. 16, No. 3, pp. 169-179, March 1973.

[DDH 72] Dahl, O. J., E. W. Dijkstra and C. A. R. Hoare, Structured Programming, Academic Press, London, 1972.

[Dem 77] Demers, A., "Generalized Left-Corner Parsing," Conference Record of the 4th ACM Symposium on Principles of Programming Languages, pp. 170-181, Los Angeles, CA, January 1977.

[DeR 71] DeRemer, F. L., "Simple LR(k) Grammars," Communications of the ACM, Vol. 14, No. 7, pp. 453-460, July 1971.

[DLP 77] Demillo, R. A. , R.J. Lipton and A. J. Perlis, "Social Process and Proofs of Theorems and Programs," Conference Record of the 4th ACM Symposium on Principles of Programming Languages, pp. 206-214, Los Angeles, CA, January 1977.

[DR 77] Druseikis, F. C. and G. D. Ripley, "Extended SLR(k) Parsers for Error Recovery and Repair," Technical Report, Departement of Computer Sciences, University of Arizona, 1977.

[FL 76] Feyock, S. and P. Lazarus, "Syntax Directed Correction of Syntax Errors," Software-Practice and Experience, Vol. 6, pp. 207-219, 1976.

[Fis 77] Fischer, C. N., "UW-PASCAL Compiler Reference Manual," Madison Academic Computing Center, University of Wisconsin-Madison, October 1977.

[FMQ 77] Fischer, C. N. , D. R. Milton and S. B. Quiring, "An Efficient Insertion-Only Error Corrector for LL(1) Parsers," Conference Record of the 4th ACM Symposium on Principles of Programming Languages, pp. 97-103, Los Angeles, CA, January 1977 (to appear in Acta Informatica).

[FL 77] Fischer, C. N. and R. J. Leblanc, "Efficient Implementation and Optimization of Run-Time Checks in PASCAL," Proceedings of the ACM Conference on Language Design for Reliable Software, pp. 19-24 Raleigh, NC, March 1977.

[FM 77] Fischer, C. N. and D. R. Milton, "Modifications to the FMQ LL(1) Error Corrector," unpublished note, June 1977.

[FTM 78] Fischer, C. N. , K. C. Tai and D. R. Milton, "Immediate Error Detection in Strong LL(1) Parsers," Technical Report #332, Computer Sciences Department, University of Wisconsin-Madison, August 1978 (also submitted to Information Processing Letters).

[GR 75] Graham, S. L. and S. P. Rhodes, "Practical Syntax Error Recovery," Communications of the ACM, Vol. 18, No. 11, pp. 639-650, November 1975.

[HB 76] Holt, R. C. and D. T. Barnard, "Syntax Directed Error Repair and Paragraphing," submitted to IEEE Transactions on Software Engineering, May 1976.

[HU 69] Hopcroft, J. E. and J. D. Ullman, Formal Languages and their Relation to Automata, Addison Wesley, Reading, MA, 1969.

[Iro 63] Irons, E. T., "An Error Correcting Parse Algorithm," Communications of the ACM, Vol. 6, No. 11, pp. 669-673, November 1963.

[Jam 72] James, L. R., "A Syntax Directed Error Recovery Method," Technical Report CSRG-13, University of Toronto, May 1972.

[JW 75] Jensen, K. and N. Wirth, "PASCAL User Manual and Report," (2nd edition) Springer-Verlag, New York, 1975.

[Knu 68] Knuth, D. E., "Semantics of Context-Free Languages," Mathematical Systems Theory, No. 2, pp. 127-145, 1968 (Correction appears in Mathematical System Theory, No. 5, p. 95, 1971).

[Lei 70] Leinius, R. P., "Error Detection and Recovery for Syntax Directed Compiler Systems," Ph.D. Thesis, Departement of Computer Sciences, University of Wisconsin-Madison, 1970.

[LRS 76] Lewis, P. M., D. J. Rosenkrantz and R.E. Stearns, Compiler Design Theory, Addison Wesley, Reading, MA, 1976.

[Mil 77] Milton, D. R., "Syntactic Specifications and Analysis with Attributed Grammars," Ph. D. Thesis, Departement of Computer Sciences, University of Wisconsin-Madison, August 1977.

[MM 78] Mickunas, M. D. and J. A. Modry, "Automatic Error Recovery for LR Parsers," Communications of the ACM, Vol. 21, No. 6. pp. 459-465, June 1978.

[Mor 70] Morgan, H. L., "Spelling Correction in Systems Programs," Communications of the ACM, Vol. 13, No. 2, pp. 90-94, February 1970.

[PD 78] Penello, T. J. and F. L. DeRemer, "A Forward Move for LR Error Recovery," Conference Record of the 5th ACM Symposium on Principles of Programming Languages, pp. 241-254, 1978.

[Pop 78] Poplawski, D. A., "Error Recovery for Extended LL-Regular Parsers," Ph.D. Thesis, Computer Science Department, Purdue University, August 1978.

[Tai 78] Tai, K. C., "Syntactic Error Correction in Programming Languages," IEEE Transactions on Software Engineering, Vol. SE-4, No. 5, pp. 414-425, September 1978.

[Wat 76] Watt, D. A., "Irons' Error Recovery in (LL and) LR Parsers," Technical Report #8, Computing Science Department, University of Glasgow, August 1976.

[Wat 77a] Watt, D. A., "The Parsing Problem for Affix Grammars," Acta Informatica No. 8, pp. 1-20, 1977.

[Wat 77b] Watt, D. A., "A Complete Definition of PASCAL by an Affix Grammar," Personal Communication, 1977.