AUTOMATIC CORRECTION OF SYNTAX ERRORS
IN PROGRAMMING LANGUAGES

TR 71 - 116

A Thesis
Presented to the Faculty of the Graduate School
of Cornell University for the Degree of
Doctor of Philosophy

by
Jean-Pierre Lévy

December 1971
BIOGRAPHICAL SKETCH

Jean-Pierre Lévy was born on January 25, 1947 in Metz, France. In June 1969, he graduated from the École Centrale des Arts et Manufactures (Paris, France) with a degree of "Ingénieur." He began graduate study at Cornell in September 1969.
A Françoise
ACKNOWLEDGMENTS

I wish to express my gratitude to Professor David Gries for his guidance, patience, and encouragements throughout the research leading to this thesis. I would also like to thank Professors R. Constable, R. Conway, and H. Morgan for serving on my special committee.

Many thanks also to Donald Johnson. Numerous discussions with him lead to invaluable clarifications of the ideas expressed in this dissertation. Steve Brown read drafts of the thesis and has provided helpful comments.

This work has been made possible by a fellowship from Cornell University and has been supported in part by National Fondation grant GJ-28176.
ABSTRACT

AUTOMATIC CORRECTION OF SYNTAX ERRORS
IN PROGRAMMING LANGUAGES

Jean-Pierre Lévy, Ph.D.
Cornell University, 1971

A very substantial fraction of the time and efforts required to develop a program is devoted to the removal of errors. In order to simplify this task, a model to automatize the correction of syntax errors is developed. It is the first model which is both formal and fairly realistic to appear in the literature.

The notion of error is defined and studied formally. Then, using this definition, a systematic error-correction process is modeled. This process makes local corrections over clusters of errors, using the context around the errors to determine the corrections and to insure that the different local corrections performed on the string do not interfere with one another. The error-correction process can be naturally embedded in many left-to-right syntax checking processes. It uses the recognizer both to detect errors and to find possible corrections.

The process has two modes: a "standard mode" used for syntax checking and an "error-correction mode" used for
determining the context of a cluster of errors and for finding all possible corrections of these errors. In the "standard mode," the syntax is checked at the same speed as if no error-correction mechanism is implemented. Thus, for programs which contain no errors, no price is paid for the presence of this mechanism. The "error-correction mode" consist of two phases: the backward move which locates the left context of the cluster, and the forward move which construct possible corrections and locates the right context of the cluster.

This process seems the most natural way to perform left-to-right syntax checking and error correction.

Some techniques for efficiently finding the range of the backward move are developed.

The formal model is not practical when using the conventional context-free description of programming languages. In order to make it more practical, the notion of bracketed context-free language is introduced and proposed as a model for the syntax of programming languages. Then, heuristic restrictions on the type of errors corrected are discussed. They may lead to a simpler process. In particular, assuming that brackets are corrected only when no other correction is possible, and that errors in deep levels of nesting (with respect to the point where the errors are detected) are neglected, it is shown how the process can be used to correct syntax errors in programming languages.
# TABLE OF CONTENTS

I. INTRODUCTION .................................................. 1

1.1 Importance of error correction and of a formal approach to the subject. ................. 1

1.1.1 Error recovery and error correction. ........................................... 1

1.1.2 Importance of a formal approach. ............................................. 2

1.2 Content and organization of the dissertation ............................................. 3

1.3 Survey of error recovery and correction techniques ................................... 5

1.4 Basic concepts and notations ................................................... 10

II. ERRORS AND ERROR CORRECTION. ........................................ 15

II.1 Introduction. ..................................................... 15

II.2 The operators r, a, & s and their extensions ......................................... 16

II.2.1 The operators r and R ....................................................... 16

II.2.2 The operators a and A ....................................................... 17

II.2.3 The operators s and S ....................................................... 20

II.3 The notion of error ................................................... 21

II.3.1 Corrections. ............................................................... 21

II.3.2 Interpretations of a string and the notion of error. ................................ 23

II.4 Error correction: a first model ........................................... 24

III. OUR MODEL FOR ERROR CORRECTION ....................................... 29

III.1 Intuitive approach .................................................. 29

III.2 Informal description of the error correction process .................................. 31

III.2.1 Overview ............................................................... 31

III.2.2 The backward move: uncertainty zones ........................................ 32
III.3 Detection of an error. 35
III.4 The error correction mode. 36
   III.4.1 The generator G 36
   III.4.2 The backward move 40
   III.4.3 The notion of beacon 44
   III.4.4 Extension of the notion of beacon 46
   III.4.5 The forward move: equivalence of prefixes 47

III.5 Algorithms for the error-correction process. 50

IV. CORRECTING REGULAR LANGUAGES 53
   IV.1 Recognizers for regular languages and equivalence of strings 53
   IV.2 Construction of the uncertainty zones 54
   IV.3 A detailed example 59
      IV.3.1 Study for \( n = 1 \) 60
      IV.3.2 Study for \( n = 2 \) 72

V. CORRECTING PROGRAMMING LANGUAGES 75
   V.1 On the class of corrected languages 75
   V.2 Parsers for programming languages 77
      V.2.1 Parsers in general 77
      V.2.2 Recognizers with one-symbol look-ahead 80
      V.2.3 RCF languages and recursive finite automata 81
   V.3 Uncertainty zones 84
      V.3.1 Problems with respect to uncertainty zones 84
      V.3.2 \( E_n(\mathcal{L}) \) generators 86
      V.3.3 Discussion 88
      V.3.4 Computation of the uncertainty zones for one symbol look-ahead parsers 89
   V.4 Error correction of "practical" programming languages 90
      V.4.1 Bracketed context-free languages 90
V.4.2 Finite uncertainty zone and bracketed RCF languages. 93
V.4.3 Bounded backward moves 95
V.5 Summary and description of the process 98

VI. SELECTING AN INTERPRETATION, MISCELLANEOUS PRACTICAL SUGGESTIONS, AND EXAMPLES. 100
VI.1 Selecting an interpretation 100
VI.2 Miscellaneous practical suggestions 102
VI.3 Examples 105
  VI.3.1 Using recursive finite automata 106
  VI.3.2 Using a SLR(1) parser 111
  VI.3.3 Discussion 119
VI.4 The error-correction process as part of a compiler 119

VII. CONCLUSION 122
VII.1 Summary of the dissertation 122
VII.2 Practicality and suggestion for future work 123

APPENDIX. COMPUTATION OF THE UNCERTAINTY ZONES FOR ONE-STEP LOOK-AHEAD 126
A.1 Transitions from truncated configurations 126
A.2 The algorithm 131
A.3 Sketch of a proof and comments 134

REFERENCES 139

vii
CHAPTER I: INTRODUCTION

I.1 Importance of error correction and of a formal approach to the subject

I.1.1 Error recovery and error correction

A very substantial fraction of the time and efforts required to develop a program is devoted to the removal of errors. Any compiler should, as much as possible, help the programmer in this chore.

Early compilers simply rejected programs as soon as an error was detected, vaguely describing the error and where it was discovered. At the present time, many compilers try to find as many errors as possible. The term error recovery is used to designate the process of determining how to continue analyzing a source program when an error is detected.

Several compilers, most notably the compilers for CORC ([CoR., 63] and [Fra., 63]), CUPL [Wal., 67], and PL/C [CoR., 70], try to "correct" all errors, generate code, and actually execute the program. The term error correction is used to designate the process which, given an incorrect program, transforms it into a correct one. The "goodness" of the process can be measured in some sense by the difference

1Throughout this chapter, material from chapter 15 of Gries's book [Gri., 71] is used. This book is a good introduction to current techniques.
between the corrected program and what the programmer actually meant. Users of these compilers find it substantially faster and easier to remove errors from programs than with conventional compilers, since, no matter how many syntactic errors, they still have a chance to find logical runtime errors.

The advantage of error correction over error recovery is twofold. First, error-correction techniques must be much more precise than error recovery; therefore they provide the programmer with a better description of his errors. Second, minor errors do not stop a program from executing, and there is a good chance it will be corrected in the right manner.

Of course, compilers become more complex with the introduction of error correction mechanisms. Compilation time might be affected (we shall show that for correct strings compilation does not need to be increased). But, we believe that the development of Computer Science should be, in part, directed toward simplifying the use of computers, relieving the user of any task computers can do. We believe this for two reasons: machines should be adapted to men and not men to machines, and the relative cost of work performed by computers, with respect to human work, is decreasing with time.

1.1.2 Importance of a formal approach

Error correction is important. This alone is sufficient to motivate a formal approach to this subject. Even though more formal automatic methods may not be directly applicable,
knowledge of these methods lends insight into how to attack the problem, and can help make the whole implementation more systematic [Gri., 71].

Furthermore, schemes for automating different parts of the compiler writer's task are currently being developed. The study of automatic error-correction mechanisms is part of this effort. Feldman and Gries [Fel., 68] state that "even a bad system would be of great value to the user."

Indeed, we feel that the results contained in this thesis justify these presumptions.

1.2 Content and organisation of the dissertation

Treatment of syntax errors in the literature is heuristic. Most often, it is recovery rather than correction which is undertaken (see next section for a survey of previous work on the subject).

In this thesis, we develop a model for error correction of syntax errors. It is the first model, which is both formal and fairly realistic, to appear in the literature. We concern ourselves with syntax because it can be formally described.

The dissertation is organized as follows:

Chapter 2: The notion of correction in formal languages is studied. Errors in a string are defined by the corrections needed to map the string into the language. A model for "global" error correction is directly derived from these notions. It is shown to be too rigid and too costly.
Chapter 3: Our model for error correction is described for formal languages which have one-way deterministic acceptors. This process makes "local" corrections over clusters of errors, using the context around the errors to determine the correction and to insure that the different local corrections performed on the string do not interfere with one another. The error-correction process can be naturally embedded in left-to-right recognizers. The parsing of correct strings is not slowed down by the presence of the error-correction mechanism. This mechanism uses the recognizer both to detect errors and to find possible corrections. The first two sections of chapter 3 are aimed at providing the reader with an informal but clear overview of the process. The remainder of the chapter formalizes the process and studies its properties.

Chapter 4: The application of our process to the correction of errors in regular languages is studied. This provides a simple and clear illustration of the process.

Chapter 5: Problems related to the correction of errors in programming languages are examined. It is shown that correcting them in the "best" way leads to an unpractical process. Simple heuristic assumptions are suggested which generally make it become practical.

Chapter 6: Some other heuristic assumptions, which may further simplify the process, are suggested. Examples of correction for an Algol-like language are described. Finally, the place of error correction in the whole compilation process is discussed.
1.3 Survey of error recovery and correction techniques

Error recovery and error correction is concerned with errors in syntax and in semantics. Logic errors cannot be detected and are therefore not subject to automatic correction.

As for semantic errors, only ad hoc recovery techniques exist. Several are described in [Gri., 71].

Misspelling can lead to syntax or semantic errors. When such errors are detected, some compilers try to determine if a spelling error actually occurred. The first work on the subject is due to Freeman [Fre., 65]. Freeman's algorithm estimates the probability that an identifier is the misspelling of another. Morgan [Mor., 70] has devised a more efficient, but less powerful, method which checks only for the following errors: one letter is wrong, one letter is missing, an extra letter is inserted, or two adjacent characters are transposed. This is the latest paper on spelling correction; it contains a list of other papers on the subject.

There are a few more papers about recovery from syntactic errors, but not many. We shall briefly describe the most characteristic ones. For other references, see [Gri., 71] and [Loi., 70].

McKeeman [McK., 70] describes a technique similar to techniques used in many bottom-up parsers. It is admittedly
primitive. It uses special characteristics of particular languages. The compiler writer gives a list of "important" symbols, like ";" and "END." When an error is detected, all input symbols are examined and discarded until one is found which is in the list. Then, the symbols on the top of the stack are successively examined and discarded until the current input symbol can legally follow what remains of the stack.

Two papers consider the problem of recovery for simple-precedence parsers [Wir., 68]. In these parsers, errors can be detected in one of two cases: the incoming input symbol is illegal, or the top of the stack does not constitute a phrase.

Wirth [Wir., 68] has a strategy for each case. When the incoming symbol is illegal, a list of insertion symbols is scanned. If some symbol of the list is legal between the top of the stack and the incoming symbol, it is inserted. Otherwise the input symbol is stacked. When a reduction cannot take place because the topmost symbols of the stack do not constitute a phrase, a table of erroneous productions is scanned comparing the right parts with the top of the stack. If a match is found, the reduction is performed and the analysis can proceed, Wirth admits that "the choice of the appropriate insertion symbols and erroneous productions requires a thorough understanding of the analysis algorithm on the part of the compiler designer, as well as a subtle
feeling to anticipate frequent misuse of the syntax." Wirth claims, however, that this method yields quite satisfying results.

Leinus [Lei., 70] approaches the same problem more systematically. His technique is fully automatic.

... the recovery algorithm disposes of errors and allows translation to continue by "reducing" a program segment to a nonterminal, despite the fact that the segment contains one or more errors. ... The recovery procedure consists of three basic steps where the three-step sequence is executed repeatedly until recovery is complete:

1. Isolate a potential phrase
2. Construct the set of possible "reductions" for the potential phrase;
3. Recover by selecting one of the nonterminal symbols in the set to replace the phrase; if the selection attempt fails, repeat from step (1).

The actual process is complex. Leinus does not justify much the choice of his algorithm, but it has the merit of being systematic.

Three more general methods exist. They are heuristic, as are the previous ones, but contrary to McKeeman's or Wirth's, no use is made of special features of a particular language.

Gries's scheme [Gri., 71] works for bottom-up parsers such that an error is detected when the next input symbol is illegal to follow the stack (for bottom-up parses, the stack contains the head of a sentential form). It tries whenever an error is detected, to insert a substring in front of the
current input symbol, such that the substring is legal in the context constituted by some stack symbols at the right-hand side, by the input symbol at the left. If no such substring exists, the current input symbol is discarded and the process repeated with the next input symbol. Admittedly, the method is just outlined and several questions left unanswered in the book. This technique has, however, been successfully used in an intuitive manner for error recovery in a compiler using transition matrices [Gri., 65].

Irons [Iro., 61] developed an error-recovery method for a top-down parser. In order to avoid backup, Irons's parsing algorithm constructs several syntax trees in parallel. At any step during the parse, one or more trees have been constructed; some branches are incomplete. An error is detected when no partial tree can be further built. Then all input symbols are successively examined and discarded until one is found which is a potential node of some incomplete branch. A terminal string is determined such that, if inserted before this input symbol, the continuation of the parse would cause this symbol to be correctly linked to the incomplete branch. The string is inserted and the parse continues. Irons's technique uses much more context than Gries's one, because the parse is top-down and contextual information is easy to extract from the incomplete trees.

LaFrance [LuF., 70 & 71] describes a recovery technique for parsers using Floyd Production Language. When an error
is detected in a state where only one next action is possible, this action is taken. Otherwise, a set of intuitive and pre-determined rules for transforming the top of the stack and a fixed number of subsequent symbols is used. Which rule to apply is determined by comparing the actual symbols with the set of symbols which "could legally be there." The process looks for a match according to a predetermined set of patterns. With each pattern is associated a transformation. For example, if the current input string is \(a_1a_2a_3\ldots\) and if \(a_2a_1a_3\ldots\) is legal in the current context, then 'a_1' and 'a_2' are permuted in the input string. Thus, this process performs transformations which are more complex than Gries's and Irons's ones.

As far as error correction is concerned, Gries states that:

Freeman [Pre., 63] is the only publication which outlines error-correction techniques in detail. It is difficult to draw any conclusions from it about general techniques for error-correction; the paper describes only correction in CORC, which is a rather simple language.

The CUPL and PL/C compilers, which we mentioned earlier, use only ad hoc techniques beside Morgan's spelling correction algorithm. No description of these techniques has been published.
Hopcroft and Ullman [Hop., 66], and Smith [Smi., 70] have studied formally error correction. These papers are highly theoretical, examine only the very specific case where errors are just substitution of symbols, and their mechanisms are very complex and time consuming. In particular, the time needed to parse a correct string is considerably greater than if a usual parser is utilized.

1.4 Basic concepts and notations

"Informally, we consider syntax to be a description of the well formed statements of the language, usually incorporating a mechanism for structural descriptions . . . " [Fel., 68]. Throughout this paper, a (formal) language $L$ is some subset of the set $\Sigma^*$ of all finite strings of symbols from a finite alphabet $\Sigma$. The description of which strings are in $L$ (syntax of $L$) is a method either for generating strings of $L$ (grammar, in particular) or for recognizing if an element of $\Sigma^*$ is in $L$.\footnote{Adapted from [Fel., 68].}

Although certain attempts to formalize all syntactic aspects of programming languages have been made [Ste., 69], the only convenient, but incomplete, models for programming languages assume that the syntax of programming languages is context-free. Therefore, we will consider only this type of syntax.
Gries's book [Gri., 71] contains most of the basic ideas about programming languages and compilers which are needed for the understanding of this dissertation. Most of the formal concepts and results we use can be found in Hopcroft's book [Hop., 69]. Our notations are similar to the one used in this text.

We shall now recall a few basic definitions.

**Grammars:** We denote a grammar by \((N, \Sigma, P, S)\). \(N\) is a finite set of nonterminal symbols. The symbols of the alphabet \(\Sigma\) are called terminal symbols. (\(N\) and \(\Sigma\) have no elements in common). The start symbol \(S\) is an element of \(N\). \(P\) is a finite set of productions (or reduction rules) of the form \(\alpha \Rightarrow \beta\) where \(\alpha\) is a string in \((N \cup \Sigma)^*\) and \(\beta\) a string in \((N \cup \Sigma)^*\).

If \(\gamma\) and \(\delta \in (N \cup \Sigma)^*\), we say that \(\gamma \beta \delta\) can be derived from or reduced to \(\gamma \alpha \delta\). We denote this by \(\gamma \alpha \delta \Rightarrow \gamma \beta \delta\). The transitive closure of the relation \(\Rightarrow\) is denoted \(\Rightarrow^*\). \(\gamma \in (N \cup \Sigma)^*\) is called a sentential form if \(S \Rightarrow^* \gamma\). The language generated by \(G\) [denoted \(L(G)\)] is the set of all sentences, that is:

\[
L(G) = \{x \mid S \Rightarrow^* x \text{ and } x \in \Sigma^*\}.
\]

There are several types of grammars. If, for every production \(\alpha \Rightarrow \beta\) in \(P\):

- \(|\alpha| > |\beta|\), then the grammar is context-sensitive
- \(\alpha = A\varepsilon N\), then the grammar is context-free (phrase-structure)
- \(\alpha = A\varepsilon N\) and \(\beta = aB\) or \(\beta = a\), with \(A\varepsilon \Sigma\) and \(B\in N\),
then the grammar is regular.

A language generated by a context-sensitive, context-free, or regular grammar is a context-sensitive, context-free, or regular language respectively.

**Recognizers:**

*Finite automata:* Regular languages are precisely the sets accepted by finite automata (finite state machines). The sets can also be described by regular expressions (expressions over finite strings using only the operations "|", ".", and "*").

A finite automaton $M$ is a system $(K, \Sigma, \delta, q_0, F)$ where $K$ is a finite, non-empty set of states

- $\Sigma$, a finite input alphabet
- $\delta$, a mapping from $K \times \Sigma$ into $K$
- $q_0 \in K$, the initial state
- $F \subseteq K$, the set of final states.

$\delta$ can be extended to mappings from $K \times \Sigma^*$ into $K$ as follows:

\[
\delta(q, \epsilon) = q \quad \forall q \in K
\]
\[
\delta(q, xa) = \delta(\delta(q, x), a) \quad \forall q \in K, \forall x \in \Sigma^*, \forall a \in \Sigma
\]

($\epsilon$ denotes the empty string).

The set of strings accepted by $M$ is denoted $T(M)$.

$$T(M) = \{ x \mid \delta(q_0, x) \in F \}$$

Given a set $S$, we denote the set of finite subsets of $S$ by
P(S). A nondeterministic finite automaton is a system
M = (K, Σ, δ, q₀, F) where K, Σ, q₀ and F have the same meaning
as for deterministic finite automata

δ is a mapping from K × Σ into \( \mathcal{P}(K) \).

δ can be extended to map \( \mathcal{P}(K) \times Σ \) into \( \mathcal{P}(K) \) as follows:

\[
\forall K' \subseteq K, \quad δ(K', a) = \{ \delta(q, a) \mid q \in K' \} \quad ∀a \in Σ
\]

δ is extended to mapping from \( \mathcal{P}(K) \times Σ^* \) into \( \mathcal{P}(K) \) as follows:

\[
\forall K' \subseteq K, \quad δ(K', ε) = K' \quad ∀ε \in Σ^*
\]

\[
\forall K' \subseteq K, \quad δ(K', xa) = δ(δ(K', x), a) \quad ∀x \in Σ^*, ∀a \in Σ
\]

Here \( T(M) = \{ x \mid δ(q₀, x) \cap F \neq ∅ \} \).

**Pushdown automaton** (pda): Context-free languages are precisely
the sets accepted by pushdown automata (whether by final state
or by empty store). A pushdown automaton \( M \) is a system
\((Q, Γ, Σ, δ, q₀, Z₀, F)\)

where
\( Q \) is a finite set of states (finite control)
\( Γ \) is a finite alphabet called the stack alphabet
\( Σ \) is a finite alphabet called the input alphabet
\( q₀ \in Q \) is the initial state
\( Z₀ \in Γ \) is the initial stack symbol
\( F \subseteq Q \) is the set of final states

δ is a mapping from \( Q \times (Σ \times \{ε\}) \times Γ \) into \( \mathcal{P}(Q \times Γ^*) \)

A configuration of a pda is a pair \((q, γ)\) where \( q \in Q \) and \( γ \in Γ^* \) (γ
represents the stack, with its leftmost symbol as top of the stack).

The PDA is deterministic if, in any configuration, only one next move is possible.

For "a" in Σ, we write: \( a:(q, Z\gamma) \rightarrow (p, \beta\gamma) \iff (p, \beta) \in \delta(q, a, Z) \),
and \( a_1a_2...a_n:(q, \gamma) \rightarrow^* (p, \beta) \iff \exists \) a sequence \( \{(q, \gamma), (q_1, \gamma_1),..., (q_{n-1}, \gamma_{n-1}), (p, \beta)\} \) such that \( a_i:(q_{i-1}, \gamma_{i-1}) \rightarrow (q_i, \gamma_i) \) for \( i = 1, ..., n-1 \).

The language accepted by final state is:

\[ T(M) = \{ w \mid w: (q_0, Z_0) \rightarrow^* (q, \gamma) \text{ for some } \gamma \in \Sigma^* \text{ and } q \in F \} \].

The language accepted by empty store is:

\[ N(M) = \{ w \mid w: (q_0, Z_0) \rightarrow^* (q, \epsilon) \text{ for some } q \in Q \} \].

Other notations:

We adopt the following convention:

Letters in Σ are represented by a, b, c, d,...

Strings in Σ* are represented by u, v, w, x, y, z,...

Stack symbols are represented by Z,...

Strings of stack symbols are represented by a, b, c, d,...

States are represented by p, q, r, s, t,...

Sets are represented by S, T, U, V, W,...

\(|x|\) denotes the length of string x; \(|W|\) denotes the size of set W.

\(\emptyset\) denotes the empty set.

The prefix of a string \( x \) is defined as:

\[ \text{prefix}(x) = \{ w \mid \exists y, \text{wy} = x \} \].

Given a subset \( S \) of \( \Sigma^* \), the definition is extended to \( S \) as follows:

\[ \text{prefix}(S) = \{ w \mid \exists y, \text{wy} \in S \} \].
CHAPTER II: ERRORS AND ERROR CORRECTION

In this chapter, we define formally what we mean by error and error correction.

II.1 Introduction

From the point of view of the syntax analyzer, a program is a string of letters (tokens, symbols). The set of all syntactically correct programs, i.e. the set of all correct strings of symbols, constitute a language. A program is incorrect if and only if it is not in the language. Syntactically incorrect programs can be viewed as programs to which symbols have been added, from which symbols have been deleted, or in which symbols have been replaced by other symbols.

All other syntax errors can be reduced to a combination of these cases. For example, transposition of two symbols can be viewed as a missing symbol and a symbol in excess.

To each of these cases (addition, deletion, and substitution of symbols) can be associated an operator. These operators can then be used as a basis for approaching the notion of error.
II.2 The operators $r, a, & s$ and their extensions

These operators replace, add, or suppress letters from strings.

II.2.1 The operators $r$ and $R$

$r$ associates with each string $x$, the set of strings which can be obtained by changing a letter in $x$.

Definition. $r$ is a mapping from $\Sigma^*$ into $\mathcal{P}(\Sigma^*)$ such that:

$$\forall x, r(x) = \{y \mid \exists u, v, a, b, \text{ such that } a \neq b, x = uav \text{ and } y = ubv\}.$$

The definition of $r$ can be extended to sets directly:

$$\forall S \subseteq \Sigma^*, r(S) = \bigcup_{x \in S} r(x).$$

The set of all strings which can be mapped into a language $L$ by changing one of their letters, is denoted $R(L)$.

Definition. $R(L) = \{x \mid r(x) \cap L \neq \emptyset\}$.

The following properties are proved in [Hop., 66].

Theorem. $R(L)$ is

a) regular if $L$ is regular

b) context-free if $L$ is context-free.

There exists a deterministic context-free language $L$, such that $R(L)$ is not deterministic.
II.2.2 The operators \( a \) and \( A \)

\( a \) associates with each string \( x \) the set of strings which can be obtained by adding a letter somewhere in \( x \).

\[
  a(x) = \{ y \mid \exists u, v, a, \text{ such that } x = uv \text{ and } y = uav \}.
\]

Similarly, \( a(S) \) is defined as \( \bigcup_{x \in S} a(x) \).

The set of all strings which can be mapped into \( L \) by addition of one letter is denoted \( A(L) \):

\[
  A(L) = \{ y \mid a(y) \cap L \neq \emptyset \}.
\]

\( A \) has the same properties as \( R \). Indeed:

**Theorem.** \( A(L) \) is

a) regular if \( L \) is regular

b) context-free if \( L \) if context-free

**Proof.** a) \( L \) regular \( \Rightarrow \) \( A(L) \) regular.

There is some finite automaton \( M \) which accepts \( L \). A non-deterministic finite automaton \( M' \) accepting \( A(L) \) can be constructed. \( M' \) consists of two sets of states: the states of \( M \), plus a copy of these states "primed." At each step during the parsing of a string, one can guess that a symbol should be inserted. In state \( s_i \), if it is guessed that \( 'a' \) should be inserted and if \( \delta(s_i, a) = s_j \), an \( \epsilon \)-transition leading to \( s_j' \) is performed. For the "primed" states, the transitions are similar to those of the corresponding states of \( M \). The accepting states of \( M' \) are the
"primed" states corresponding to the accepting states of $M$.

Formalization: Let $M = (S, \Sigma, \delta, s_0, F)$ be a finite automaton which accepts $L$ ($M$ may be non-deterministic). Let $M'$ be the non-deterministic finite automaton $(S', \Sigma, \delta', s'_0, F')$ where:

- if $S = \{s_0, s_1, \ldots, s_n\}$ then
  \[ S' = \{s_0, s_1, \ldots, s_n, s'_0, \ldots, s'_n\} \]
- $F' = \{s'_i | s_i \in F\}$
- $\delta' : S' \times \Sigma \rightarrow P(S')$ such that
  \[ \forall s_i, \forall a \quad \delta'(s_i, a) = \delta(s_i, a) \]
  \[ \delta'(s'_i, a) = \{s'_j | s_j \in \delta(s_i, a)\} \]
  \[ \delta'(s'_i, \epsilon) = \{s'_j | \exists a, s_j \in \delta(s_i, a) \cup \delta(s_i, \epsilon)\} \]
  \[ \delta'(s'_i, \epsilon) = \{s'_j | s_j \in \delta(s_i, \epsilon)\}. \]

Claim: the set $T(M')$ of strings accepted by $M'$ is identical to $\Lambda(L)$

i) $T(M') \subseteq \Lambda(L)$:

$$ x \in T(M) \Rightarrow \exists s' \in F', s' \notin \delta'(s_0, x). \text{ There are no transitions from a "primed" state to a "non-primed" state,}$$
and all paths from "non-primed" to "primed" states are through $e$-transitions. Hence $\exists u, v, x = uv$ and $\exists s_i, s'_j \in S'$ such that:

$s_i \in \delta'(s_o,u)$, $s'_i \in \delta'(s_i,e)$, $s' \in \delta'(s'_i,v)$

$\Rightarrow s_i \in \delta(s_o,u)$, $s \in \delta(s_j,v)$ and $\exists a$, $s_j \in \delta(s_i,a)$

hence $s \in \delta(s_o,uv) \Rightarrow uv \in L \Rightarrow x \in A(L)$.

ii) $A(L) \subseteq T(M')$:

$x \in A(L) = \exists u, v, a \ni x = uv \& uv \in L$

$\exists s_i, s_j$ and $s \ni s_i \in \delta(s_o,u)$

$s_j \in \delta(s_i,a)$

$s \in \delta(s_j,v)$ \& $s \in F$

$\Rightarrow s_i \in \delta'(s_o,u)$, $s_j \in \delta'(s_i,e)$, $s' \in \delta(s'_j,v)$ and $s \in F'$

hence $s' \in \delta'(s_o,uv) \Rightarrow x = uv \in A(L)$.

This ends the proof of a).

b) $L$ context-free $\Rightarrow A(L)$ context-free can be proven by changing the finite control of the recognizer in a similar way.

**Theorem.** There exists a deterministic context-free language $L$, such that $A(L)$ is not deterministic (A(L) is ambiguous).
Proof. Let \( L = \{ a^k b^k c^m d^m \mid k, m \geq 1 \} \cup \{ a^k b^m c^m d^k \mid k, m \geq 1 \} \).

\( L \) is deterministic context-free. Let us assume that \( A(L) \) is deterministic. Then its intersection by a regular set is deterministic [Hop., 69]. Hence \( A(L) \cap \# a^* b^* c^* d^* \) is deterministic. But, this set is

\[
\{ a^k b^k c^m d^m \mid k, m \geq 1 \} \cup \{ a^k b^m c^m d^k \mid k, m \geq 1 \}
\]

which is ambiguous, therefore not deterministic [Hop., 69].

This contradiction implies that \( A(L) \) is not deterministic. In fact, \( A(L) \) is clearly ambiguous.

II.2.3 The operators \( s \) and \( S \)

\( s(x) \) is the set of all strings of \( \Sigma^* \) which can be obtained by suppressing a letter somewhere in \( x \):

\[
s(x) = \{ y \mid \exists u, v, a, \text{ such that } x = uav \text{ and } y = uv \}.
\]

\( s \) is extended to operate on set of strings as above. The set of all strings which can be mapped into \( L \) by suppressing one of their letters is called \( S(L) \):

\[
S(L) = \{ x \mid s(x) \cap L \neq \emptyset \}.
\]

\( S \) has the same properties as \( R \) and \( A \).

Theorem. \( S(L) \) is

a) regular if \( L \) is regular

b) context-free if \( L \) is context-free.
Proof. a) $L$ regular $\Rightarrow S(L)$ regular:

Idea: The idea is similar to the proof of the same theorem for $A(L)$. $M'$ constructed from two copies of $M$. Here, the guess is that a letter should be ignored. The schematization looks like:

\[
\begin{array}{c}
M' \\
\begin{array}{c}
\text{a} \\
\text{s}_i \\
\text{s}_j \\
\end{array}
\end{array}
\quad - b - \quad \begin{array}{c}
\text{a} \\
\text{s}_i \\
\text{s}_j \\
\end{array}
\]

The formalization is similar, as is the proof of $b$.

Theorem. There exists a deterministic context-free language $L$, such that $S(L)$ is not deterministic.

Proof. Let $L = \{w \neq w^t | w \in \Sigma, w^t$ is the transpose of $w, \theta \not\in \Sigma\}$. Let us assume that $S(L)$ is deterministic.

$\Rightarrow L' = S(L) \cap \Sigma^* = \{ww^t | w \in \Sigma^*\}$ is deterministic, but $(ww^t)$ is not deterministic [Hop., 69] $\Rightarrow$ contradiction.

Hence $S(L)$ is non-deterministic.

11.3 The notion of error

11.3.1 Corrections

As explained at the beginning of this chapter, the basic errors to which all errors can be reduced are letters missing, substituted for, or in excess. Therefore in each
case, the corresponding action of addition, replacement, or deletion of symbols shall be called a correction. The set of all strings which can be obtained from $x$ by one correction is noted $c(x)$. It is defined as:

$$c(x) = a(x) \cup r(x) \cup s(x).$$

The set of all strings which can be mapped into $L$ by one correction is noted $C(L)$.

$$C(L) = \{ x \mid c(x) \cap L \neq \emptyset \} = A(L) \cup R(L) \cup S(L).$$

The operator $c^n$ corresponds to $n$ corrections,

$$c^0(x) = x$$
$$c^n(x) = c(c^{n-1}(x)) \quad \text{for } n \geq 1.$$ 

Similarly,

$$C^0(L) = L$$
$$C^n(L) = C(C^{n-1}(L)) \quad \text{for } n \geq 1.$$ 

The sets of all regular languages and of all context-free languages being closed under union, the following theorem can be derived from the theorems about $A$, $R$, and $S$.

**Theorem.** $C^n(L)$ is a) regular if $L$ is regular
b) context-free if $L$ is context-free.

There exists deterministic context-free languages such that $C^n(L)$ is not deterministic (for $n \geq 1$).
II.3.2 Interpretations of a string and the notion of error

To define the notion of error, we shall proceed in a way similar to Hamming [Ham., 50] in his discussion of error-correcting codes.

The distance between two strings $x$ and $y$ is defined as the minimum number of corrections needed to transform $x$ into $y$ (or vice versa).

The basic problem of error correction is, given a string $x$ which is not in $L$, to find which string of $L$, $x$ stands for. In other words, the programmer intended to write a string $y$ of $L$; he made some errors and wrote $x$ instead of $y$. We know $x$, what is $y$? Or more precisely, what is $y$ most likely to be?

We postulate that $x$ stands for one of the nearest strings of $L$. These strings are called interpretations of $x$ (the term correction is reserved to designate a basic operation on strings). The word interpretation is used to convey the idea that we are not really sure that any of these strings was intended in place of $x$. We just make this assumption, because it is the most natural one.

Definition. Given $x \notin L$, $y \in L$ is an interpretation of $x$ iff $y \in c^k(x)$ and $\forall i < k$, $c^i(x) \cap L = \emptyset$.

If the distance between $x$ and its interpretations is $k$, $x$ is said to contain $k$ errors.
There may be several interpretations of $x$. Any of these is a candidate to represent $x$. An interpretation must eventually be selected. The selection algorithm can be directed by experimental considerations such as statistical analysis, a learning process, etc. Our concern is to find the set of interpretations. We shall assume the existence of an adequate selection function (some ideas about the form of such a function are given in chapter VI).

The errors in $x$ are defined a posteriori by the corrections needed to map $x$ into $L$. They even depend on the interpretation selected as meaning of $x$.

II.4 Error correction: a first model

Assume we adopt the following strategy:
- if $x$ contains at most $n$ errors, find a "good" interpretation of $x$,
- if $x$ contains more than $n$ errors, reject it.

The set of all strings which will be corrected by this process is

$$\bigcup_{i=0}^{n} C^i(L).$$

The theorem of section II.3.1 implies that this set is regular if $L$ is regular, context-free if $L$ is context-free. If $L$ is a deterministic context-free language, this set may be nondeterministic (as a matter of fact, it happens in many important cases).
Given a recognizer \( M \) for a language \( L \), we want to construct a generator \( G \) which generates all possible interpretations of an input string which contains at most \( n \) errors. Its construction will now be described informally for \( L \) regular, context-free, or as a matter of fact any member of an A.F.L.\(^1\)

\( M \) is a finite automaton, a pushdown automaton, or more generally a one-way balloon automaton. It consists of a finite control and, possibly, of an auxiliary storage (none for a finite automaton, a push-down store for a pda, a "balloon" for a balloon automaton). The finite control of \( G \) consists of \( n+1 \) copies of the finite control of \( M \); its auxiliary storage is a copy of the storage of \( M \). \( G \) is nondeterministic. Transitions from one copy of the finite control of \( M \) to another correspond to guesses that a correction is needed. The copies are numbered from 0 to \( n \), where the copy number corresponds to the number of corrections performed. To each transition in the finite control is associated an output which is either:

- the symbol read, if it is assumed to be correct (the transition is the same as the one in \( M \))
- nothing (\( \epsilon \)) if the symbol read is suppressed
- the character inserted if there is an addition
- the symbol replacing the symbol read if a replacement takes place.

\(^1\)On A.F.L. and balloon automata, see [Hop., 67] and [Gin., 69].
When the whole input has been read, a string with a minimum number of corrections is selected from the set of output strings.

This construction is illustrated by the following figure.

\[ M : \]

\[ G : \]

Transitions between boxes correspond to corrections.

The construction of \( G \) is described formally in the next chapter.

Discussion of this model

The complexity of \( G \) is close to the complexity of
the recognizer of

\[ \bigcup_{i=0}^{n} C^i(L) \]

(up to a term proportional to the length of the input string for generating and selecting the output strings). Therefore, if \( L \) is regular, \( G \) works in a time proportional to the length of the string since

\[ \bigcup_{i=0}^{n} C^i(L) \]

is regular. That is, if the length of the input string is \( L \), the time \( T(L) \) needed to process this string is bounded by \( \alpha_n \), for some constant \( \alpha_n \).

A priori, a linear time bound seems reasonable. The size of the constant depends not only \( \hat{a} \) \( n \) but also \( \hat{a} \) \( \hat{a} \) the implementation. Therefore any discussion about it can only be very informal. Assume \( M \) has \( s \) states. \( G \) could be implemented as a deterministic machine, but the number of states would be \( s \times 2^{ns} \) which is likely to be unpractical. If \( G \) is kept in a nondeterministic form, \( \alpha_n \) is proportional to the maximum number of states in which \( G \) can be at any moment. An upper bound for this number is \( ns+1 \) (one state corresponding to 0 corrections and \( s \) states corresponding to \( i \) corrections, \( i = 1, 2, \ldots, n \)). Thus, \( \alpha_n \) can grow at most linearly with \( n \). We conjecture that in practical cases this bound is reached.
If \( L \) is context-free,
\[
\bigcup_{i=0}^{n} c^i(L)
\]
is context-free. If \( L \) is deterministic, it can be recognized in linear time, whereas the smallest known time bound for an ambiguous nondeterministic context-free language is proportional to \( \lambda^3 \) [Hop., 69; Ear., 70].
\[
T(\lambda) \leq a_n \lambda^3
\]
Here also, \( a_n \) is likely to be growing linearly with \( n \).

Let us now discuss the model. If one wants to accept a substantial number of errors, the run time of the process is long. Furthermore, with this model the maximum number of errors in the strings corrected is a constant. It does not depend of the length of the string.

Of course, we do not want to correct strings which contain too many errors. The cause for this great number of errors is likely to require human attention. But, by "too many errors" we do not mean a constant number of errors, independent of whether the string is long or short. What we mean is that the string should be corrected if errors are not "too dense."

In the next chapter, we develop a model for error correction which remedies these flaws.

---

1 On the other hand, one does not know any context-free language which cannot be recognized in linear time by some specific recognizer [Hop., 69]. However, as just said, general syntax-directed recognizers such as Earley's have a cubic time bound.
CHAPTER III: OUR MODEL FOR ERROR CORRECTION

Our model for error correction is now explained, first intuitively and next formally.

III.1 Intuitive approach

Any reasonable error-correction process should be such that the amount of work required is proportional to the number of errors in the string. In particular, if a string is correct, it should be processed in a time not substantially greater than the time needed to parse the string.

This remark directs us toward a process which has two modes: a standard mode to be used when no error is detected, and an error-correction mode to be used otherwise.

The standard mode uses a habitual parsing technique. The error correction should be done from left to right in order to be able to perform it, more or less, in parallel with this parsing technique, using the parser to find possible corrections.

It will be supposed that errors occur in clusters, in which there are no more than \( n \) errors. The clusters are "far enough" one from another to be easily distinguished. This vague notion will be made more precise later. It can be illustrated by the following figure:
(↑ points to the errors)

no more than n errors in a cluster

We would like to use an error-correction mode only in the vicinity of these clusters of errors. We will proceed by successive approximations. The errors will be corrected cluster after cluster, from left to right, in a way such that the corrections chosen for one cluster do not affect the correction of the others.

In order to explain how this is done, we introduce the concept of equivalence of prefixes. Given a language L, a string x is said to be a prefix of L if there exists a string y (suffix) such that xy is a string of L. Two prefixes are equivalent if any suffix which can follow one can follow the other.

Before we describe the process, let us note that we will deal only with deterministic languages, that is languages for which there exists deterministic recognizers. Indeed, all practical languages are deterministic.
III.2 Informal description of the error-correction process

III.2.1 Overview

(Certain terms which will be used now, have not yet been defined, but their general meaning should be obvious.)

First, the string is scanned from left to right in the "standard mode," checking only that the part read is a correct prefix. Whenever the prefix is no longer correct, we say that an error has been detected. Then one switches to the first phase of the "error-correction mode."

This phase consists of a backward move (from right to left) finding the least number of characters to the left of the last point scanned in which a correction may be needed. This substring is called the left context of the cluster of errors.

The second phase consists of a forward move constructing all possible interpretations (with at most n corrections) of the prefix read until all interpretations are equivalent. The substring between the point where the error is detected and the point where all interpretations are equivalent is called the right context of the cluster of errors.

Finally, one of these interpretations is selected and replaces the prefix. Then the process returns to its "standard mode" of operation, resuming the checking of correctness.

1This is a slight misuse of language since, in fact, only the existence of one or more errors has been detected and not the error itself.
Equivalence of all interpretations insures that which one is selected is indifferent to the correction of later clusters of errors. We will briefly discuss this selection in chapter VI. If there exists no interpretation of the prefix (with at most \( n \) corrections), the correction process aborts.

A string which can be corrected is said to have \textit{at most} \( n \) errors locally (with respect to \( L \)).

The process is illustrated by the following figure:

\[ x : \longrightarrow \]  
usual left-to-right scan

\[ x : \begin{array}{c}
\begin{array}{c}
\text{an error is detected} \\
\rightarrow \\
\end{array}
\end{array} \]

\[ x : \begin{array}{c}
\begin{array}{c}
\text{backward move to find the} \\
\text{left context of the cluster} \\
\text{of errors} \\
\leftarrow \\
\end{array}
\end{array} \]

\[ x : \begin{array}{c}
\begin{array}{c}
\text{forward move constructing all potential} \\
\text{interpretations until they are all equivalent} \\
\rightarrow \\
\end{array}
\end{array} \]

\[ x : \begin{array}{c}
\begin{array}{c}
\text{select correction \( d \) of \( bc \) and return to the "standard mode"} \\
\rightarrow \\
\end{array}
\end{array} \]

\[ x \] is the string being processed, \( b \) and \( c \) the left and right context of the errors.
Note: In the remainder of this dissertation, we will not say explicitly "for a given bound \( n \) on the local number of errors and with respect to a language \( L \)." For example, when we speak of the interpretations of a prefix, we refer implicitly to the language \( L \) and the maximum number of errors \( n \) with which we are currently concerned. The letters 'L' and 'n' will always be used to mean this language and this bound.

III.2.2 The backward move: uncertainty zones

We now justify and describe still informally, but more precisely, the backward move. Given a string

\[ a_1 a_2 \ldots a_j a_{j+1} \ldots \]

assume an error is detected at \( a_{j+1} \). As just explained, we want to construct all possible interpretations of

\[ a_1 a_2 \ldots a_j \ldots a_{j+k} \]

starting with \( k \) equal to one, for increasing values of \( k \), until all possible interpretations of \( a_1 \ldots a_{j+k} \) are equivalent.

A first solution would be to run the generator \( G \) (defined in section II.4.1) from the beginning of the string. This is extremely time consuming and is precisely what we want to avoid.

We will prove (III.4.2) that in fact it is only necessary to run \( G \) on some substring \( a_i \ldots a_{j} \), where \( i \)
may be close to $j$. The computation of $i$ involves the knowledge of the sets of states in which $G$ was when reading $a_j, a_{j-1}, \ldots, a_i$ (recall that $G$ is nondeterministic).

It is a vicious circle since this information cannot be computed beforehand. To obtain it, one has to run $G$ on the prefix. However, the computation of $i$ can be done using at each step a set which includes the set of states in which $G$ is at the same step. This computation yields a value $i'$ ($1 \leq i' \leq i$). Hopefully, $i'$ will be close to $i$ (this is not always the case).

At a given step, the only information that we have is the state of the recognizer $M$. So, if $M$ is in state $q$ after having scanned $a_1 \ldots a_i$, we will use, in place of the set of states of $G$ at this step, the set of all states in which $G$ could be after having read any string which get $M$ in $q$. This set of states of $G$ is called the uncertainty zone associated with state $q$. We shall see that in some cases uncertainty zones can be computed beforehand on some form or another. Let us indicate that for programming languages these zones are not directly computable. Some more assumptions on the nature of errors will be needed. This problem is studied in chapter V.

The above ideas are formally studied in section III.4.2. An example of an uncertainty zone is given for a simple regular language in IV.3.
In the remainder of this chapter, we define formally and study the notions introduced in these first two sections. Their application to regular and programming languages is studied in subsequent chapters.

III.3 Detection of an error

Definition. An error is detected at 'a' in the string $x$ if $x \in \text{prefix}(L)$ and $xa \notin \text{prefix}(L)$.

Usually, recognizers detect errors by having no possible move when reading $a$.

The fact that an error is detected at some point does not mean that the error is close to that point. Indeed:

Theorem. There exists a regular language such that the distance between the point where the error is detected and the corrections is unbounded.

Proof. Let $L$ be $\{0^n#\} \cup \{50^n$$\}$. Consider the string $x = 0^k$$$. An error is detected at the first "$.$" The unique string of $L$ which is at a minimum distance from $x$ is $50^k$$$. The corresponding correction is to change the first "$#" into "$$." The distance between the point where the error is detected and the correction is $k$ which can be as large as desired.
III.4 The error-correction mode

III.4.1 The generator $G$

We deal with languages for which there exist a one-way deterministic recognizer. Assume $M$ is such a recognizer for the language $L$. We want to construct the generator $G$ which given a prefix $u$ of some string generates all strings $v$ where $v$ is a prefix of $L$ and $v$ is at a distance at most equal to $n$ from $u$.

We will have to consider the states of $M$. They should not be confused with the states of the finite control of $M$. (For example, if $M$ is a pda, the states of $M$ are pairs of a state of the finite control and of a stack configuration. In this case, $M$ has an infinite number of states.) $M$ can be described by a system of the form:

$$M = (K, \Sigma, \delta, q_0, F)$$

where:

- $K$ is the set of states of $M$
- $\Sigma$ is the input alphabet
- $\delta$ is a transition function which maps $K \times \Sigma$ into $K$
- $q_0 \in K$ is the initial state
- $F \subseteq K$ is the set of accepting states.

As usual, the definition of $\delta$ is extended to mappings from $K \times \Sigma^*$ into $K$ as follows:
\[ \delta(q, e) = q \]

\[ \delta(q, xa) = \delta(\delta(q, x), a) \text{ for each } x \text{ in } \Sigma^* \text{ and } a \text{ in } \Sigma. \]

A string \( x \) is in \( L \) iff \( \delta(q_0, x) \in F \).

Furthermore, we shall assume that \( \delta(q_0, x) \) is defined iff \( x \in \text{prefix}(L) \).

The generator \( G \) has been described in section II.4.1. It is now defined formally as a system of the form:

\[ G = (K', \Sigma, \{q_0\}, \delta', F') \]

where:

- \( K' = K_0 \cup K_1 \cup \ldots \cup K^n \)

  where \( K_i = (p_i, q_i, r_i, \ldots) \) is the \( i \)th copy of the states \( K = \{p, q, r, \ldots\} \) of \( M \)

- \( \delta' \) maps \( K' \times \Sigma \) into \( (\Sigma(K) \times \Sigma^*) \). The interpretation of \((p, w)\) being in \( \delta(q, a) \) is that \( G \) in state \( q \) with input symbol \( a \) may, as one possible choice of move, enter state \( p \) and generate string \( w \)

- \( F' = F_0 \cup F_1 \cup \ldots \cup F^n \) where \( F_i \) is the image of \( F \) in \( K_i \) i.e. \( F_i = \{p_i \mid p \in K_i\} \).

\( \delta' \) is derived from \( \delta \) as follows:
\[ \delta'(q^i, a) = \begin{cases} (p^i, u) & \text{one of the following cases holds} \\
1) \delta(q, a) = p, i = 2 \text{ and } u = a \\
2) \exists b, b \neq a, \delta(q, b) = p, i = 2+1, \\
i \leq n, \text{ and } u = b \\
3) \exists v, v \neq a, |v| = m, \delta(q, va) = \\
p, i = 2+m, i \leq n \text{ and } u = va \\
4) p^i = q^{i+1}, i \leq n \text{ and } u = \varepsilon. \end{cases} \]

The cases in the definition of \( \delta' \) mean:

1) no correction
2) the symbol \( a \) is replaced by \( b \)
3) the string \( v \) is inserted before \( a \)
4) \( a \) is suppressed.

Remarks:

1) The function \( \delta' \) is expressed in a way such that there are no \( \varepsilon \)-moves if \( \delta \) has no \( \varepsilon \)-moves. This is more convenient to work with.
2) In (4) one can add the condition "\( \exists b, b \neq a, \delta(q, b) \) is defined" since it is useless to delete a symbol which is the only one having the right to be there. (We will use this remark in all our examples.)
3) This definition of \( \delta' \) does not permit insertion of symbols after the last letter of the input. However, in practice the last symbol is an end marker.
which is not subject to correction. Below, it shall always be assumed that L ends with such a symbol, noted "I".

The definition of \( \delta' \) is extended to the domain \( K' \times \Sigma^* \) as follows:

\[
\delta'(q^i, \epsilon) = \{(q^i, \epsilon)\}
\]

\[
\delta'(q^i, xa) = \{(p^j, w) \mid w = w_1w_2, \exists x^k, (x^k, w_1) \in \delta'(q^i, x) \text{ and } (p^j, w_2) \in \delta'(x^k, a)\}.
\]

This mapping can be further extended to the domain \( \mathbb{P}(K') \times \Sigma^* \) as follows:

\[
\delta'(S, x) = \bigcup_{p \in S} \delta'(p, x).
\]

Note that when \( M \) is a finite automaton, \( G \) is a generalized sequential machine, when \( M \) is a pushdown automaton, \( G \) is a pushdown transducer (on g.s.m. and p.d.t. see [Gin., 66]).

Notation:

\[
\delta_1(S, x) = \{p \mid (p, w) \in \delta'(S, x)\}.
\]

\(\delta_1(S, x)\) is the set of all first elements of the pairs of \( \delta'(S, x) \).

\[
\delta_2(S, x) = \{w \mid (p, w) \in \delta'(S, x)\}.
\]

\(\delta_2(S, x)\) is the set of all second elements of the pairs of \( \delta'(S, x) \).
Now, following the idea developed for the first error-correction model, we define the set of potential interpretations of a prefix $u$ of a string $x$ as being the set of prefixes of $L$ which distance from $u$ is less than or equal to $n$. This set is $\delta_L(q_0, u)$.

III.4.2 The backward move

We now develop formally the ideas introduced in III.2.2. Assume a string $a_1a_2 \ldots a_ja_{j+1} \ldots$ is being processed. An error has just been detected at $a_{j+1}$. We want to construct the set of potential interpretations of $a_1a_2 \ldots a_j \ldots a_{j+k}$ for $k = 0, 1, 2, \ldots$ until all potential interpretations of $a_1 \ldots a_{j+k}$ are equivalent.

A possible solution is to go all the way back, to the beginning of the string and feed the whole prefix through $G$. But, when discussing the first model, we objected precisely to this need of processing the whole string through a device which is very time consuming. We shall now show that we do not always need to go back to the beginning of the string.

First, let us imagine that $G$ runs over $a_1a_2 \ldots a_j$. After reading $a_1 \ldots a_k$ ($k \leq j$), $G$ is in a set of states $V_k$ of the form $\{p_0^k\} \cup \{q^m \mid 1 \leq m \leq n\}$ where $p$ is the state in which $G$ would be after reading the same prefix $(p^k = \delta(q_0, a_1 \ldots a_k))$. This can be deduced directly from the fact that only one string corresponds to $a_1 \ldots a_k$ with zero corrections—it is $a_1 \ldots a_k$ itself!
The states of $V_j$ are generated from the states of $V_{j-1}$ by a $\delta'$-transition under $a_j$. But some of the states of $V_{j-1}$ may give an undefined transition under $a_j$. Let $W_{j-1}$ be the set of states of $V_{j-1}$ which are effectively mapped into $V_j$:

$$W_{j-1} = \{ p^m | p^m \in V_{j-1} \text{ and } \delta^*_1(p^m, a_j) \cap V_j \neq \emptyset \}.$$ 

Generalizing, we define $W_k$ for $k = j-2, j-3, \ldots, 1$ as:

$$W_{k-1} = \{ p^m | p^m \in V_{k-1} \text{ and } \delta^*_1(p^m, a_k) \cap W_k \neq \emptyset \}.$$ 

$W_{k-1}$ is the set of states of $V_{k-1}$ which have an image in $V_j$ under the $\delta'$-transition over $a_k \ldots a_j$.

Clearly, as $V_k, W_k$ is of the form $(p^0) \cup \{ q^m | 1 \leq m \leq n \}$. Therefore, if $W_k$ contains only one element, it must be in $V^0$, and, as seen easily, $W_{k-1}, W_{k-2}, \ldots, W_1$ contain only one element.

Let $W_i$ be the last set $W$ with only one element, i.e.:

$$|W_0| = 1, |W_1| = 1, \ldots, |W_i| = 1, |W_{i+1}| > 1.$$ 

Assume $W_{i+1} = (p^0)$, then $V_j = \delta^*_1(p^0, a_{i+1} \ldots a_j)$.

This means that the set of potential interpretations of $a_1 \ldots a_j$ is $a_1 a_2 \ldots a_i \delta^*_1(p^0, a_{i+1} \ldots a_j)$. Thus, to find all potential interpretations of this prefix, we need to run $G$ only on $a_{i+1} \ldots a_j$.

$a_{i+1} \ldots a_j$ is the left context of the cluster of
errors. Hopefully, this context will be small. However from the proof of the theorem in section II.3, we can deduce that in certain cases the context is the whole prefix.

The above computation of the left context involves the knowledge of the states in which \( G \) is at each step, that is \( V_L \). But to compute this states one has to run \( G \) on \( a_1 \ldots a_k \) which is precisely what we are trying to avoid; we want to use only information about the behavior of \( M \). After having read \( a_1 \ldots a_k \), \( M \) is in state \( q_k \) and \( G \) in states \( V_L \). \( V_L \) clearly depends not only on \( q_k \) but also on the path followed to reach \( q_k \). From the knowledge of only \( q_k \), what can we say of \( V_L \)? We can find a set which includes \( V_L \). This set is the set of states of \( K' \) in which \( G \) could be in after having read any string \( x \) which puts \( M \) in state \( q_k \). This set is said to be the uncertainty zone associated with \( q_k \). Formally:

**Definition.** The uncertainty zone associated with a state \( q \) of \( M \), is the set \( U_q \) such that

\[
U_q = \{ s^i \mid \exists x, \delta(q_0, x) = p \text{ and } s^i \in \delta_1(q_0, x) \}
\]

Clearly,

\[
V_L \subseteq U_{q_k}
\]

since \( V_L = \delta_1(q_0, a_1 \ldots a_k) \). It is also easy to show that, for any \( q \), \( U_q \) is of the form \( (q^0) \cup \{ p^m \mid 1 \leq m \leq n \} \).

Now, instead of computing the sets \( W \), we compute sets
T which are defined similarly:

\[ T_{k-1} = \{ p^m \mid p^m \in U_{q_{k-1}} \text{ and } \delta_i^l(p^m, a_{j}) \cap T_i \neq \phi \} \text{ and } T_j = U_{q_j}. \]

If the sets \( U_{q_j} \) can be computed beforehand, one can compute the sequence of sets: \( T_j, T_{j-1}, \ldots \) Clearly \( W_k \subseteq T_k \) for \( k = j, \ldots, l \). Thus, \( |T_k| = 1 \) implies that \( |W_k| = 1 \).

So we compute \( T_j, T_{j-1}, \ldots \) until some \( T_i \) has a size of one. This \( i' \) is certainly less than or equal to \( i \), that is \( a_i' \ldots a_j \) is larger than the left context of the error. It is, therefore, sufficient to run \( G \) on this substring, which is computable from the knowledge of only \( q_j, \ldots, q_i' \) if the uncertainty zones are computable.

In brief, the algorithm for the backward move, using uncertainty zones, is the following:

1. \[ T \leftarrow U_{q_j} ; \quad l \leftarrow j \]
2. \[ \text{if } |T| = 1 \text{ then start forward move} \]
3. \[ l \leftarrow l - 1 \]
   \[ T \leftarrow \{ p^m \mid p^m \in U_{q_k} \text{ and } \delta_i^l(p^m, a_{k+1}) \cap T \neq \phi \} \]
4. \[ \text{go to } [1] \]
One of the problems we will have to study is how to compute the uncertainty zones. We shall see that it is easy if $L$ is regular, more difficult if $L$ is context free (in this case some more assumptions will be needed).

III.4.3 The notion of beacon

Here we study a way to speed up the previous algorithm. Let us examine step [2]. It performs:

$$T = \{ p^m \mid p^m \in U_{q_k} \quad \text{and} \quad \delta'_1(p^m, a_{k+1}) \cap T \neq \emptyset \}.$$  

The algorithm stops if $|T| = 1$. We know that prior to this computation $T \subseteq U_{q_{k+1}}$. Let $T'_k$ be the set:

$$T'_k = \{ p^m \mid p^m \in U_{q_k} \quad \text{and} \quad \delta'_1(p^m, a_{k+1}) \cap U_{q_{k+1}} \neq \emptyset \}.$$  

$T'$ is the set of all states of $U_{q_k}$ which are predecessor of a state of $U_{q_{k+1}}$. Clearly $T \subseteq T'$, therefore if $T'$ has only one element ($q^0$), $T$ has also only one element. That is, without being obliged to compute the sequence of $T'$s, we can affirm that the backward move must stop at or before $a_k$ if the transition

$$q_k \xrightarrow{a_{k+1}} q_{k+1}$$

is such that $|T'| = 1$.

Such a transition marks a point in the string before which no correction is needed in any case (if the prefix $a_1 \ldots a_k$ is a valid prefix of $L$). One can build a priori
a list of the transitions which have this property. They are called "beacons." (Note that since $N$ is deterministic, a transition can be described by the pair $(q,a)$ of the state before the transition and of the input symbol.)

**Definition.** A transition $(q,a)$ is called a **beacon** if and only if the set

$$\{ p^m \mid p^m \in U_q \text{ and } \delta'_1(p^m, a) \cap U_q \neq \emptyset \}$$

has only one element $\ast$

We now show that this is equivalent to say that, besides $q^0$, no state of $U_q$ has an image over $a$ (by $\delta'_1$). This property is easier to check.

**Theorem.** The transition $(q,a)$ is a beacon if and only if

$$\forall p^m \in U_q \setminus, p^m \neq q^0 \Rightarrow \delta'_1(p^m, a) = \emptyset \ast$$

**Proof.** Assume $(q,a)$ is a beacon. $p^m \in U_q$ means:

$$\exists x, \delta(q_0, x) = q \text{ and } p^m \in \delta'_1(q_0, x).$$

Now assume $\exists t^k, t^k \in \delta'_1(p^m, a)$ for some $p^m$ in $U_q$.

Then $\delta(q_0, xa) = \delta(q, a)$ and $t^k \in \delta'_1(q_0, xa)$ which means that $t^k \in U_{\delta(q, a)}$.

This implies that $(q,a)$ is not a beacon which contradicts the assumption. Hence:

$(q,a)$ is a beacon $\Rightarrow (\forall p^m \in U_q \setminus, p^m \neq q^0 \Rightarrow \delta'_1(p^m, a) = \emptyset)$.

The reciprocal is immediate $\ast$
The above remarks lead to another algorithm for the backward move. The computation is faster and simpler than the one introduced in the last section. However, the backward move may be larger with, consequently, more work to do during the forward move. The trade-off may be favorable, especially if there are numerous beacons in the language.

This new algorithm for the backward move is simply: "go back to the last beacon."

Examples will be studied in the following chapters. We shall see that beacons exist because of redundant features of the language. Also, the number of beacons tends to decrease as $n$ increases, because the size of the uncertainty zones increases with $n$. Therefore, we shall now generalize the notion of beacon in order to have beacons in $L$ even for larger $n$'s.

III.4.4 Extension of the notion of beacon

To define beacons, we considered only one transition. Now, let us consider a sequence of transitions:

$q \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \ldots \xrightarrow{a_{m-1}} q_{m-1} \xrightarrow{a_m} q_m$.

This sequence is denoted $(q, a_1 a_2 \ldots a_m)$. It will be called an $m$-beacon if it has the same property as a beacon, i.e., $q$ is the only state of $U_q$ which is a predecessor of a state of $U_{q_m}$. 
Definition. The sequence of \( m \) transitions \((q, a_1 ... a_m)\) is called an \( m \)-beacon if and only if:

\[
\forall p^j \in U_q, \quad p^j \neq q^0 \Rightarrow \delta_1^i(p^j, a_1 ... a_m) = \phi
\]

Beacons, as defined previously, and 1-beacons are identical. It can be shown, as done for beacons, that if \((q, a_1 ... a_m)\) is an \( m \)-beacon and if this sequence of transitions appears in a string \( ... a_1a_2 ... a_m ... \) then the backward move must stop before or directly at the left of \( a_1 \).

III.4.5 The forward move: equivalence of prefixes

The forward move is performed by starting \( G \) at the beginning of the left context of the cluster of errors (or at the left of the last beacon, if beacons are used for the backward move). If at that point \( M \) was in state \( q \), \( G \) is started in state \( q^0 \). The forward moves stops when all interpretations are equivalent. We will therefore define precisely what we mean by "equivalence of prefixes" and study properties of this equivalence relation.

Definition. Two prefixes \( u \) and \( v \) of \( L \) are said to be equivalent with respect to \( L \) iff

\[
\forall z, \quad uz \in L \leftrightarrow vz \in L
\]

The equivalence of these prefixes is noted:

\[
u \equiv_L v \quad \text{(or } u \equiv v \text{ if there is no ambiguity)}.
\]
This definition is not constructive. In fact:

**Theorem.** Given a context-free grammar $G$, it is undecidable whether $u \in L(G) \lor \cdot$

**Proof.** It is undecidable whether two context-free grammars generate the same language [Hop., 69]. We show below that if the equivalence of two strings were decidable, then the equivalence of grammars would be decidable too. Therefore, equivalence of prefixes is undecidable.

Let $G_1$ and $G_2$ be two grammars. Without loss of generality, we assume that the two sets of nonterminals are disjoint.

$$G_1 = (N_1, \Sigma, P_1, S_1) \text{ and } G_2 = (N_2, \Sigma, P_2, S_2)$$

Let $S$, $\#$, and $\$ be symbols which are not in $N_1 \cup N_2 \cup \Sigma$.

The grammar $G$,

$$G = (N_1 \cup N_2 \cup \{S\}, \Sigma \cup \{\#, \$\}, P_1 \cup P_2 \cup \{S \rightarrow \#S_1, S \rightarrow \$S_2\}, S),$$

generates $L = \#L(G_1) \cup \$L(G_2)$.

Now, $\# \in L \iff \forall x, \#x \in L \iff \$x \in L$

i.e. $\forall x, x \in L(G_1) \iff x \in L(G_2)$

i.e. $L(G_1) = L(G_2)$ Q.E.D. \cdot

This theorem means that there is no algorithm which, when given $L$, $x_1$, and $x_2$, decides whether $x_1 \in L x_2$.

Fortunately, we shall see that it is possible to find sufficient conditions for the equivalence to be true.
Whether the above theorem is true for deterministic context-free languages is an open question, since the problem of equivalence of deterministic context-free grammars is open [Hop., 69].

We now give a sufficient condition for the equivalence of prefixes.

**Theorem.** If \( u \) and \( v \) are such that \( \delta(q_0, u) = \delta(q_0, v) \) then \( u \equiv v \).

**Proof.**

\[
\forall z, \delta(q_0, uz) = \delta(\delta(q_0, u), z) = \delta(\delta(q_0, v), z) = \delta(q_0, vz)
\]

Hence, \( uz \in L \Rightarrow \delta(q_0, uz) = \delta(q_0, vz) \in F \Rightarrow vz \in L \) Q.E.D.

**Corollary.** If \( (p^i, u) \in \delta'(q_0, x) \) and \( (p^j, v) \in \delta'(q_0, x) \) for some \( x \) then \( u \equiv v \).

**Proof.** The proof follows directly from the previous theorem, if one notes that \( (p^i, u) \in \delta'(q_0, x) \) implies that \( \delta(q_0, u) = p \).

In the same way, \( \delta(q_0, v) = p \).

This important result means that two interpretations, which put \( G \) in two states which are copies of the same state of \( M \), are equivalent.

**Remark:** During the forward move, some selection may possibly be performed among equivalent prefixes, before all prefixes are equivalent. For example, if \( (p^0, x) \) and \( (p^i, u) \) are both in \( \delta'(q_0, x) \) then one can clearly eliminate \( u \).
We have now all the elements needed to describe precisely the whole error-correction process.

### III.5 Algorithms for the error-correction process

The first algorithm described uses beacons (1-beacons).

During the error-correcting forward move, the set \( V \) of all states of \( G \), at a certain step, is constructed. Three pieces of information are attached to each state of \( G \): the generated string, the position of the last beacon on this string, and the state corresponding to it.

To determine the equivalence of prefixes, we use the sufficient condition: all states of \( G \) are copies of the same state of \( N \) (see III.4.4).

The general organization of the algorithm is:

1. **Initialization**
2. **Scan string from left to right until an error is detected**
3. **Backward move**
4. **Forward error-correcting move**
   1. **Initialization** (\( V \): set of states of \( G \) plus attached information)
   2. **Compute new set of states of \( G \)**
   3. **If this set is empty then abort (more than \( n \) errors in a cluster)**
   - else if all strings generated are equivalent, select one of them and insert it in the input string in place of the original one; go to [1]
   - else read the next input character and go to [3.2]
We present now a more precise version of this algorithm. The string parsed is $x$ which is of the form $a_1a_2 \ldots a_i \ldots \perp$. 'b' indicates the last beacon encountered, 'p' the state corresponding to it.

[0] $q \leftarrow q_0; i \leftarrow 1; b \leftarrow 0; p \leftarrow q_0$

[1] while $\delta(q,a_i)$ is defined do
  begin if $a_i = \perp$ then stop
  if $(q,a_i)$ is a beacon then begin $b \leftarrow i - 1$ ; $p \leftarrow q$ end
  $q \leftarrow \delta(q,a_i)$
  $i \leftarrow i + 1$
  end

[2] $i \leftarrow b ; q \leftarrow p$

[3.1] $V = \{(q^0, \epsilon, i, p)\}$

[3.2] $W = \phi ; i \leftarrow i + 1$
  for all $(t^j, u, b', p') \in V$ do
    begin if $(t,a_i)$ is a beacon then
      begin $b' \leftarrow i - 1$ ; $p' \leftarrow t$ end
      $W = W \cup \{(s^k, uv, b', p') \mid (s^k, v) \in \delta(t^j, a_i)\}$
    end

[3.3] if $|W| = 0$ then abort
  if all states in $W$ are copies of the same state of $M$
  then begin select $(s^k, u, b', p')$ among all elements of $W$
    $q \leftarrow s ; b \leftarrow b' ; p \leftarrow p'$ ; go to [1]
  end

$V = W$ ; go to [3.2]
An important property of this algorithm is that if one wants to perform a (left-to-right) translation of the program, everything before the last beacon can be translated since this part of the string is never changed.

Now, we sketch the implementation of the process using two other methods for the backward move.

First, let us examine the implementation of the technique using $m$-beacons. The principle of the algorithm is the same as previously but one must examine the state of the machine $M$, $m$ steps before the current one as well as the previous $m$ input symbols. The bigger $m$ is, the more work there is.

The last algorithm uses uncertainty zones for the backward move. This requires that the sequence of states of $M$ during the left-to-right scan be recorded. The algorithm for the backward move is described in III.4.2. During the forward error-correcting move, the sequence of states of $G$ must also be recorded. A possible implementation is to keep this sequence in a tree form (see example in IV.3.2).

In the next chapters, the application of the method to regular and context-free languages is studied.
CHAPTER IV: CORRECTING REGULAR LANGUAGES

Regular languages are a first approximation to programming languages. The process described in the previous chapter can be applied directly to such languages, once the uncertainty zones can be computed. The application of our error-correction scheme to regular languages provides a very clear illustration of this scheme.

IV.1 Recognizers for regular languages and equivalence of strings

There is, up to an isomorphism, a unique deterministic finite automaton with a minimum number of states, which accepts a given regular language. Finite automata are machines of the form described in the previous chapter with a finite number of states (in particular, recall that it is required that \( \delta(q_0, x) \) be defined if and only if \( x \) is a prefix of \( L \)).

For regular languages, equivalences of prefixes is decidable. Indeed:

Theorem. Given a regular language \( L \) and its minimal recognizer

\[
M = (K, \Sigma, \delta, q_0, F),
\]

two prefixes \( u \) and \( v \) are equivalent with respect to \( L \)
iff:

53
\[ \delta(q_0, u) = \delta(q_0, v) \]

**Proof.** We proved in III.4.4 that
\[ \delta(q_0, u) = \delta(q_0, v) \Rightarrow u \equiv_L v. \]

Now, assume that \( u \equiv_L v \). It means that
\[ \forall z, u z \in L \iff v z \in L. \]

This equivalence relation is exactly the one used to construct and prove the uniqueness of the minimal finite automaton accepting \( L \) (consequences of the Myhill-Nerode's theorem, see [Hop., 69]).

Hence, \( u \equiv v \Rightarrow \delta(q_0, u) = \delta(q_0, v) \).

Thus, if \( M \) is the minimal finite automaton recognizing \( L \), the forward move as described in the last chapter stops exactly when all interpretations are equivalent.

IV.2 Construction of the uncertainty zones

The error-correction algorithm has been fully described in the previous chapter, except for the construction of the uncertainty zones.

Let us recall that the uncertainty zone associated with a state \( q \) of \( M \) is the set of states in which \( G \) could be after having read any string which puts \( M \) in state \( q \).

\[ V_q, U_q = \{ p^m \mid \exists x, \delta(q_0, x) = q \text{ and } p^m \in \delta'(q_0, x) \}. \]

Note that \( \delta' \) is the transition function of the generator \( G \) constructed from \( M \), as described previously. Here \( G \)
is a generalized sequential machine (on g.s.m. see [Gin., 66] or [Hop., 69]).

Without loss of generality, we assume that \( M \) is minimal. We construct all the uncertainty zones by induction. To each \( q \), we associate a set \( W_q \) which is a subset of \( U_q \). Each induction step increases the size of the \( W \)'s until \( W_q = U_q \), for all \( q \).

We showed that the \( U_q \) was of the form

\[
\{ q^0 \} \bigcup \{ p^m \mid 1 \leq m \leq n \}
\]

So, we start with \( W_q = \{ q^0 \} \) for all \( q \). At each step

\[
\delta^i(W_p,a)
\]

is computed for each state \( p \) and input symbol \( a \). Assume that

\[
q = \delta(p,a),
\]

then

\[
\delta^i(W_p,a)
\]

is a subset of \( U_q \), therefore \( W_q \) is replaced by

\[
W_q \cup \delta^i(W_p,a).
\]

This computation is illustrated by the following figures:
before induction step

induction step: compute image of $W_p$ by $\delta_1$ over $a$

after induction step
The algorithm stops when no $W_q$ grows. The $W_q$'s are then identical to the $U_q$'s.

This algorithm is now precisely described. A detailed proof follows.

Algorithm which computes the uncertainty zones:

for all $q \in K$ do $W_q \leftarrow \{q^0\}$ ; flag $\leftarrow$ true

while flag $=$ true do

begin flag $\leftarrow$ false

for all $a \in \Sigma$ do for all $p \in K$ do unless $\delta(p,a)$ is undefined do

B: begin $q \leftarrow \delta(p,a)$

$T \leftarrow \delta_1(W_p,a)$

unless $T \subseteq W_q$ do begin flag $\leftarrow$ true

$W_q \leftarrow W_q \cup T$

end

end

end

Proof of correctness.

We claim that, when the above algorithm terminates, the $W_q$'s are the uncertainty zones. To prove it, we use techniques similar to the ones introduced by Floyd [Flo., 67] and Hoare [Hoa., 69].
1) The algorithm terminates:

The \( W ' s \) are subsets of \( K' \) which is finite. After each execution of the body of the loop, either the size of at least one \( W_q \) has increased or the computation stops. Since the size of the \( W ' s \) is bounded (and is an integer), the computation must stop.

2) When the algorithm terminates, \( W_q = U_q \) for all \( q \):

i) for all \( q \), \( W_q \subseteq U_q \):

To say \( W_q \subseteq U_q \) is equivalent to say

\[ \forall t^i \in W_q, \exists x \text{ such that } \delta(q_0, x) = q \text{ and } t^i \in \delta_1(q_0, x). \]

When the block \( B \) is entered for the first time,

\[ \forall q, W_q = \{ q_0 \} \]

which satisfies the above condition. We shall show that if \( \forall q, W_q \subseteq U_q \) when block \( B \) is entered, then this property is still true after the execution of \( B \). Hence, it will be true after any execution of \( B \), in particular when the computation terminates.

So, assume that \( \forall q, W_q \subseteq U_q \) and execute \( B \) for some \( p \) and \( a \).

\( q = \delta(q, a) \).

\( \forall r, r \neq q, W_r \) is unchanged.

Now, \( \forall s^j \in T, t^i \in W_p, s^j \in \delta_1(t^i, a) \).

The assumption implies:

\[ \forall s^j \in T, \exists x \text{ such that } s^j \in \delta_1(q_0, xa) \text{ and } p = \delta(q_0, xa). \]
That is \( T \subseteq U_q \). Therefore the new \( W_q \) ('old' \( W_q(U_I) \) is a subset of \( U_q \).

ii) for all \( q \), \( W_q \supseteq U_q \)

The proof is done by contradiction.

Assume that there exists some \( q \) and some \( p_i \), such that \( p_i \in U_q \) and \( p_i \notin W_q \). The algorithm stopped, therefore:

\[ (**') \forall r, a, \delta'_1(W_r, a) \subseteq W_t \] \( \text{ (with } t = \delta(r, a) \) \]

but \( p_i \in U_q \Rightarrow \exists x, q = \delta(q_o, x) \) and \( p_i \in \delta'_1(q_o, x) \).

Let \( x = a_1 a_2 \ldots a_k \) and \( \delta(q_o, a_1) = q_1 \)

\[ \delta(q_1, a_2) = q_2 \]

\[ \ldots \]

\[ \delta(q_{k-1}, a_k) = q_k \]

\[ (**') \Rightarrow \delta'_1(W_q, a_1 a_2 \ldots a_k) \subseteq W_q \]

That is \( \delta'_1(q_o, x) \subseteq W_q \).

Hence \( p_i \in W_q \), which contradicts the assumption. Q.E.D.

IV.3 A detailed example

The strings of the language, which we are using for our example, consist of zero or more declarations followed by one or more assignment statements:
L = (type id ( , id)* ;)* id + id (+ id)* ( ; id + id (+ id)* )* \\

Note: '(', and ')', are metasymbols.

This language is not too far from some programming languages (a very simple Fortran without parenthesized expressions).

The minimal finite automaton which recognizes L is:

---

IV.3.1 Study for \( n = 1 \)

For \( n = 1 \), the transition table of \( G \) is as follows:

(Note: \( t = \text{type} \), \( i = \text{id} \), \( c = '\,' \))

also, the states of \( K^0 \) are denoted by \( \{1, \ldots, 7\} \),

those of \( K^1 \) by \( \{1', \ldots, 7'\} \).
<table>
<thead>
<tr>
<th>type</th>
<th>id</th>
<th>;</th>
<th>;</th>
<th>;</th>
<th>;</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1'</td>
<td>(2', t)</td>
<td>(4', i)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2'</td>
<td></td>
<td>(3', i)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3'</td>
<td></td>
<td>(2', c)</td>
<td>(1', ;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5'</td>
<td></td>
<td>(6', i)</td>
<td></td>
<td></td>
<td>(5', +)</td>
<td></td>
</tr>
<tr>
<td>6'</td>
<td></td>
<td></td>
<td>(7', ;)</td>
<td></td>
<td>(5', +)</td>
<td>(8', 1)</td>
</tr>
<tr>
<td>7'</td>
<td></td>
<td>(4', i)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
63

Notes: As already explained, in our examples we select arbitrarily a string among the ones with a minimum number of corrections (rather: the selection is done by choosing what we think subjectively to be the "best" string). During the error-correcting forward move, whenever several prefixes are equivalent, we keep only one of them. This is not, by all means, the only or even the "best" policy but it is the simplest.

Hence, in the previous table when two corrections led to the same state, we kept only one. For example, 6 under id yields among others (6', e) and (6', *i); we kept (6', *i) (from the "semantic" point of view that it is more likely that '*' be missing than an identifier be in excess).

Note also that, as signaled in III.4.1, whenever an input symbol is the only legal one which can appear when \( M \) is in a certain state, we do not need to guess a correction. For example, if '*' is the input symbol while in state 4, one goes directly to 5 because this transition will have to be eventually performed in any case.

Uncertainty zones

The algorithm described in IV.2 yields:

\[
U_1 = \{1, 2', 2'\} \\
U_2 = \{2, 1', 3', 4'\} \\
U_3 = \{3, 4'\} \\
U_4 = \{4, 1', 2', 3', 6'\}
\]
\[ U_5 = \{ 5, 6', 7' \} \]
\[ U_6 = \{ 6, 4' \} \]
\[ U_7 = \{ 7, 5', 6' \} \]
\[ U_8 = \{ 8 \} \]

**Beacons (1-beacons):**

One finds: (1, type) (3, c) (3, ;) (4, +) (6, ;) (6, +) and obviously (6, !).

**Remark:** Most of these beacons are transitions of the form (p,a) such that there exists no transition (q,a) with q \( \neq p \), that is (p,a) is the only transition over a. For \( n = 1 \), this is a general property as we shall see now.

**Theorem.** A sufficient condition for the transition (p,a) to be a beacon, for \( n = 1 \), is that it is the only transition over a in \( M \) (i.e. q \( \neq p \) \( \Rightarrow \) \( \delta(q,a) \) is undefined).

**Proof.** Consider \( r^1 \notin U_p \). According to the definition of the uncertainty zones, \( \exists x, (q_0,x) = p \) and \( (r^1,w) \in \delta'(q_0^0,x) \) for some \( w \) obtained from \( x \) by one correction. Now,

\[ \delta'(r^1,a) = \{ t^1 \mid t = \delta(r,a) \} \]

since at most one correction is allowed. But the transition over a is unique which implies that \( \delta(r,a) \) is undefined \( (r^1 \notin p^1 \) since \( p^0 \) is in \( U_p \)). Hence \( \forall r^1 \notin U_p \), \( \delta'(r^1,a) = \emptyset \) which implies, according to the theorem in III.4.3, that (p,a) is a beacon.
Let us note that, although the transitions under ':' are not unique, they are beacons. Intuitively, it is because these transitions are "far" one from the other in the transition diagram and only one of them can occur in a given context.

An example of correction

The string which is analyzed is:

\[ \text{id}, \text{id}; \text{type id}, \text{id + id + id}, \text{id}; \text{id + id + id + id}; \text{id + id} \]

As shown below, the error-correction process yields:

\[ \text{type id, id; type id; id + id + id + id; id + id}; \text{id + id}; \text{id + id} \]

The following figures describe the process. Each page exhibits the correction of one cluster of errors (here each cluster has only one error since \( n = 1 \)). The string is written from top to bottom, instead of from left to right. Beside each symbol is written the state in which \( M \) is after having read the symbol. The backward move is performed using beacons. The right-hand side of each figure shows the error-correcting forward move. Each row shows the states of \( G \) at this step. To each state of \( G \) are associated the symbols added to the generated string. When two states are equivalent, one of them is selected arbitrarily. In each figure, the corrected prefix is substituted for the original one.
<table>
<thead>
<tr>
<th>String</th>
<th>State</th>
<th>Forward error-correcting move</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>1</td>
<td>(4,1) (1',ε) (2',t) (3',t1)</td>
</tr>
<tr>
<td>,</td>
<td>4</td>
<td>(4',ε) (5',ε) (2',c)</td>
</tr>
<tr>
<td>id</td>
<td></td>
<td>(6',i) (3',i)</td>
</tr>
<tr>
<td>;</td>
<td></td>
<td>(7',;') (1',;') (2',t)</td>
</tr>
</tbody>
</table>
\[
\begin{align*}
\text{id} & \quad 4 \quad 5 \quad 6 \\
+ & \quad 5 \quad (5, *) \quad (6', c) \quad (7', ;) \\
\text{l} & \quad (8', i\perp) \quad \text{equivalent} \quad (9', \perp)
\end{align*}
\]
IV.3.2 Study for $n = 2$

For $n = 2$, the uncertainty zones are:

- $U_1 = \{1', 2', 2', 4'', 5'', 7''\}$
- $U_2 = \{2, 1', 2', 4', 1'', 3'', 4'', 5''\}$
- $U_3 = \{3, 4', 1'', 2'', 4'', 5'', 6''\}$
- $U_4 = \{4, 1', 2', 3', 6', 1'', 6''\}$
- $U_5 = \{5, 6', 7', 1'', 2'', 3'', 4'', 6'', 7''\}$
- $U_6 = \{6, 4', 3'', 4'', 5'', 7''\}$
- $U_7 = \{7, 5', 6', 1'', 4'', 5''\}$
- $U_8 = \{8\}$

There are no 1-beacon, besides the obvious one: $(6, \downarrow)$. There are two 2-beacons $(3, ;t)$ and $(6, +i)$ plus, clearly, $(5, i\downarrow)$.

In the following figures, we show how

$$\text{id} \quad \text{id} ; \text{type id} \quad \text{id} ; \text{id} + + \text{id} + \text{id} \downarrow$$

is interpreted (using uncertainty zones for the backward move) as

$$\text{type id} \quad \text{id} ; \text{type id} \quad \text{id} ; \text{id} + \text{id} + \text{id} \downarrow$$

or as:

$$\text{type id} \quad \text{id} ; \text{type id} \quad \text{id} \quad \text{id} ; \text{id} + \text{id} + \text{id} \downarrow$$

Note that the backward move using 2-beacons would be larger for the second cluster of errors.
CHAPTER V: CORRECTING PROGRAMMING LANGUAGES

The application of our technique to the syntax of programming languages is now studied.

We first show that the language of correctable strings may not be context-free even though the target language is. Then, different types of parsers for programming languages are examined. Problems related to the computation of the uncertainty zones and some possible solutions are discussed. An algorithm is described which, under certain conditions, generates the uncertainty zones. Finally, using the notion of representation of a language by a system of equations over sets, we study the bracketed structure of programming languages and its consequences with respect to error correction.

V.1 On the class of corrected languages

The set of strings which can be corrected using the global error-correction process introduced in chapter II is a context-free set if the language $L$ is context-free (see II.4.2). We shall now see that it is not the case for our (local) error-correction process. This means that, given a grammar $G$ for $L$, there is no way to generate a context-free grammar $G'$ for the set of correctable strings. Hence, there is no way to expand $G$ to include error-correcting
productions which would simulate our process.

Theorem. There exists a context-free language, such that the
set of strings which can be corrected into this language,
with at most one local correction, is a context-sensitive
set.

Proof. Let \( L \) be \( \{a^m b^m c^n d^m | m, n > 1\} \cup \{a^m b^n c^n d^m | m, n \geq 1\} \).

Let \( C(L) \) denote the set of strings which can be corrected
into \( L \) with at most one (local) correction. We shall study

\[
C' = C(L) \bigcap \{da^i b^j c^k d^m | i, j, k \geq 1\}.
\]

Consider a string \( da^i b^j c^k d^m \). An error is detected when
the first symbol is read. The error-correction process starts
building two interpretations of the prefixes of the input
string. They are of the form \( \#... \) and \( \$... \). While read-
ing \( a \)'s and \( b \)'s, these interpretations cannot be equivalent.

Now, consider the prefix \( da^i b^j c \). Its two interpretations are: \( da^i b^j c \) and \( Sa^i b^j c \). The possible suffixes are \( c^{i-1} d^m \) and
\( c^{j-1} d^m \) respectively. One of the following cases holds:

i) \( i \neq j \). The two interpretations are not equivalent.

When \( \# \) is read, \( G \) is unable to proceed since one correc-
tion has already been performed. The input string cannot be
corrected, and is therefore not in \( C' \).

ii) \( i = j \). The two interpretations are equivalent.

Either \( \#a^i b^i c \) or \( \$a^i b^i c \) is selected, substituted to
the prefix, and the standard syntax checking resumed. An
error is detected when reading '#'. Some backward move is performed, then G is used again to construct interpretations. The string will eventually be corrected if i = k+1 or k+2 ('#' suppressed or changed to 'c'). This means that:

$$C' = \{ da^i b^i c d c^k d \downarrow | k \geq 1 \text{ and } i = k+1 \text{ or } k+2 \}.$$  

C' is context-sensitive (not context-free). C' is the intersection of C(L) by a regular set, therefore C(L) is context-sensitive.

V.2 Parsers for programming languages

V.2.1 Parsers in general

There are numerous syntax-directed parsing techniques based on a context-free description of the language. The most important ones are described or mentioned in a state-of-the-art survey by Feldman and Gries [Fel., 68].

All these parsers have a finite control and a push-down stack. A state (configuration) of the parser is a pair (q,γ) where q is a state of the finite control and γ describes the content of the stack (γ is a string of stack symbols; its left-most character represents the top of the stack). We denote the set of states of the finite control by Q and the stack alphabet by Γ. A recognizer M can be represented formally as in III.4.1. The set K of states of this recognizer is Q×Γ*.
Thus, parsers can have an infinite number of states. Practical parsers are generally not push-down automata. In particular, the transition function may have another form: some bounded look-ahead of the input string may be performed, a bounded number of top-most symbols of the stack may be examined and changed. These parsers can, however, always be expressed as PDA's.

**Notation**

If over the input symbol \( a \), \( M \) goes from state \((p,a)\) to state \((q,\beta)\), we write it:

\[
a : (p,a) \rightarrow (q,\beta).
\]

If \( M \) goes from \((p,a)\) to \((q,\beta)\) over the string \( x \), we denote it:

\[
x : (p,a) \rightarrow^* (q,\beta).
\]

Whenever there is no ambiguity on the transition function, we use the symbols '→' and '→*' instead of '→' and '→*'.

For example, if \( M \) is a PDA, \( \delta \) maps \( Q \times \Sigma \times \Gamma \) into \( Q \times \Gamma^* \) such that \( \delta(p,a,Z) = (q,\alpha) \) means \( a : (p,Z\gamma) \rightarrow (q,\alpha\beta) \). In the next section, we shall see an example of a transition function expressed differently.

The generator \( G \) can be constructed from \( M \) using the method described in section III.4.1 (note that the set \( X \) of states of \( M \) is \( Q \times \Gamma^* \)). Here also the transition function of
G is denoted $\delta'$ and G is not deterministic. If $M$ is a push-down automaton, G is a push-down transducer [Gin., 66]. The states of G are of the form $(p, a)^i$ where $0 \leq i \leq n$. Equivalently, they can be denoted $(p^i, a)$. Thus, the set of states of G is denoted $Q' \times \Gamma^*$ where
\[ Q' = Q^0 \cup Q^1 \cup \ldots \cup Q^n \]
and
\[ Q^i = \{ p^i : p \in Q \}. \]

We require that $M$ detects incorrect prefixes. Some parsing techniques do not have that power (e.g., precedence techniques [Wir., 66], transition matrices [Gri., 68], ...). However, many recent and efficient techniques are capable of detecting incorrect prefixes (e.g., LR(k) [Knu., 65], SLR(k) [DeR., 71], recursive finite automata [Tix., 67], ...). In any case, it seems reasonable, if one tries to correct errors in the "best" way, to require the parser to be able to detect errors as early as possible.

Finally recall that equivalence of prefixes is undecidable for context-free languages. As discussed in III.4.5 we will use a stronger condition, that is: two prefixes will be considered equivalent if and only if they put $M$ in the same state. The following results are, therefore, centered on a given recognizer for the language rather than on the language itself. This is one consequence of the fact that there is no general algorithm to construct a canonical (minimal) recognizer for a given context-free language.
V.2.2 Recognizers with one-symbol look-ahead

Recognizers for context-free languages can be expressed as pda's. This representation, useful for obtaining theoretical results and for proofs, is not a close model of the actual behavior of practical parsers. There are two main differences between usual recognizers and pda's. Generally, recognizers examine a bounded amount of symbols from the top of the stack and they perform some look-ahead on the input string.

We will now introduce a formal model for such parsers, when the look-ahead is no more than one input symbol. We study only one symbol look-ahead for two reasons. First, practical implementations use no more than one symbol look-ahead. Second, for larger look-ahead, the symbols looked at must be considered as a part of the state of the recognizer; this makes the study much more difficult.

Recognizers with one-symbol look-ahead are very similar to the classical pda's. In fact, only the form of their transition function is different in order to model more closely the behavior of actual recognizers.

δ maps $Q \times \Sigma \times \Gamma^*$ into $Q \times \Gamma^* \times \{0,1\}$ where 0 indicates that the reading head does not move on the input string (i.e. an $\epsilon$-move is performed), 1 indicates that the head moves to the next input symbol. There is look-ahead because the recognizer looks at the next input symbol in
order to decide whether to read it or to perform an ε-move.

It is convenient to adopt the convention that

\[ a : (q, a) \rightarrow (p, \beta) \]

denotes going from \((q, a)\) to \((p, \beta)\) by performing all possible ε-moves and then the transition over \(a\). That is, the state of the recognizer is examined before any ε-move is performed. No generality is lost by using this convention.

V.2.3 RCF languages and recursive finite automata

We now sketch a model for programming languages which emphasizes certain characteristic features of these languages. It will be used to illustrate and discuss some of our further results. The corresponding recognizer is well suited to error correction.

Ginsburg and Rice [Gin., 62] have shown that every context-free language \(L \subseteq \Sigma^*\) can be defined by a system of equations over sets of strings of the form:

\[ \begin{align*}
A_1 &= r_1 \\
A_2 &= r_2 \\
& \vdots \\
A_k &= r_k
\end{align*} \]

where the \(r_i\)'s are regular expressions over \(\Sigma \cup A_1 \cup \ldots \cup A_k\) using only the operations '.', '∪', and '⁎'. The \(A_i\)'s are subsets of \(\Sigma^*\). They are called variables. The system
has only one solution such that the size of all \( A_i \)'s is minimum. The set \( A_1 \) of such a solution is the language \( L \) defined by the system.

It is interesting to represent context-free languages by such a system of equations, using a "minimal" number of variables because these equations describe, in a sense, what is regular and what is context-free in the language. Generally, each equation corresponds to a level of nesting.

Tixier [Tix., 67] describes transformations of the system which can be used to reduce the number of variables. The definition of Algol 60 is not completely context-free because of the distinction between boolean expressions and arithmetic expressions. If a covering of these expressions is used, Algol can be described by a system of 3 equations:

\[
\begin{align*}
\text{program} & = \text{block} \\
\text{block} & = r_1 \\
\text{expression} & = r_2
\end{align*}
\]

where \( r_1 \) is a regular expression over \( \Sigma \cup \text{block} \cup \text{expression} \) and \( r_2 \) is a regular expression over \( \Sigma \cup \text{expression} \).

Under certain conditions formalized by Tixier, a deterministic parser can be derived directly from the system. To each regular expression corresponds a finite automaton whose input symbols are elements of \( \Sigma \) and variables (variables denote automata). Automata call one another recursively.
The recognizer was informally introduced by Conway [Con., 63]. Our brief description stems form [Fel., 68]. Let us first note that we assume that no expression can begin with a variable. Tixier shows that the system can always be put in such a form.

The left-to-right recognizer consists of the finite automata plus a push-down stack. In our formalism, the state of the finite control is the current state $q_i$ of the current automaton $A_i$. The stack symbols are pairs of a name of an automaton and a state of the same automaton. The transition function is defined as follows:

a) if in the current automaton there is a transition from the current state $q_i$ to state $p_i$, the stack is unchanged, the state $p_i$ becomes the current state, and the next input symbol is scanned;

b) if in the current automaton there is a transition from $q_i$ to $p_i$ over $A_j$ and if $A_j$ may begin with the input symbol, then the names of the current automaton and of $p_i$ are pushed onto the stack, the new current state is the initial state of the new current automaton $A_j$; an $\epsilon$-move is performed;

c) if neither (a) nor (b) occur and if the current state is a final state of the current automaton, the current state and automaton are changed to the one specified by the top of the stack which is
popped up; an ε-move is performed.

a) corresponds to a transition inside automaton \( A_1 \).
b) corresponds to a call of automaton \( A_j \).
c) corresponds to a return from automaton \( A_1 \).

Tixier's condition - separability - when satisfied insures that it is possible to determine unambiguously from the current state and from the input symbol if another finite automaton should be called, and if so, which one. Thus the recognizer is deterministic. Languages which satisfy these conditions are called regular context-free (RCF).

The recognizer is top-down, each automaton corresponding to a prediction. Note that look-ahead is used to determine which move to perform and whether to read a new symbol.

V.3 Uncertainty zones

V.3.1 Problems with respect to uncertainty zones

The definition of uncertainty zones (III.4.2) applied to parsers for context-free languages is:

\[
U_{q, \alpha} = \{(p^i, \beta) \mid \exists x \, x : (q_0, z_0) \vdash \alpha \rightarrow (q, \alpha) \\
\text{and } x : (q_0^0, z_0) \vdash \delta^1 \rightarrow (p^i, \beta)\}.
\]

In general, there exists an infinite number of configurations (states of the recognizer) since the stack is unbounded. It is therefore impossible to have a table which contains the uncertainty zones associated with each state.
If the notion of uncertainty zone is to be used, we need an effective and efficient way for constructing dynamically the (finite number of) uncertainty zones required for the backward move.

However, there exist context-free languages for which the size of the uncertainty zones, for any of their recognizers, is unbounded. That is, for any bound $N$, there exists $(q, a)$ such that $|U_{q,a}| > N$.

We shall first prove this theorem, then discuss its consequences.

**Theorem.** There exists a context-free language $L$, such that, for any of its (left-to-right) recognizers, the size of the uncertainty zones is unbounded.

**Proof.** Let $n = 1$ and $L$ be the language generated by the following grammar:

- $B \rightarrow \text{begin } B \text{ end}$
- $B \rightarrow (E)$
- $E \rightarrow (E)$
- $E \rightarrow B$
- $E \rightarrow \text{var}$

Assume the prefix $\text{begin \hspace{1cm} begin \hspace{1cm} ...} \hspace{1cm} \text{begin}$ is being parsed $N + 1$ times.

Among other possible corrections, a parenthesis may be inserted between two $\text{begin}$. There are $N$ such interpretations of the prefix. Clearly, no two such interpretations are equivalent,
which implies that \( G \) is in at least \( N+1 \) different states after having read the prefix. If \((q, a)\) is the state of the recognizer \( M \) after having read the prefix, each of these \( N+1 \) states of \( G \) are part of the uncertainty zone associated with \((q, a)\). Hence, \[ |U_{q,a}| \geq N + 1 > N \]

This example is not far fetched; it represents the skeleton of some programming languages. This means that, in general, it may be difficult to construct the uncertainty zones. Additional assumptions on the nature of possible errors may have to be made in order to bound the size of these zones.

In brief, we have to solve two problems:

1) there is an infinite number of configurations,

2) the size of the uncertainty zones may be unbounded (even infinite for some "unnatural" cases).\(^1\)

Before looking at these problems in the general case, let us discuss a type of recognizer for which uncertainty zones can be easily constructed.

V.3.2 \( E_n(x) \) generators

We are looking for a fast way to reconstruct the uncertainty zones. For the time being, it will be assumed that the second problem has been solved, i.e. that the uncertainty zones are bounded.

\(^1\)For example, consider \( L = \{a^n b^n | n \geq 1\} \cup \{a^n | n > 1\} \) and the uncertainty zone associated with the configuration in which \( M \) is after having read some prefix \( S a^k \). The uncertainty zone associated with this configuration is infinite.
Under certain conditions, we had like to construct the uncertainty zone associated with a configuration \((q, \gamma)\) just from the knowledge of \(q\) and of the \(k\) top-most symbols of the stack \(\gamma\). Let \(\gamma'\) denote these \(k\) symbols, \(\xi\) the remainder of the stack \((\gamma = \gamma'\xi)\). We shall call \((q, \gamma')\) a truncated configuration.

Why is such an idea reasonable and interesting? In a very rough sense, it means that in such recognizers the uncertainty zone is only function of the "last" symbol scanned. In some parsing techniques, content of the stack and level of nesting are closely related. Thus, the above supposition means that only the inner levels of nesting can be erroneous.

**Definition.** A generator \(G\) is said to be \(E_n(\xi)\) if and only if the following condition holds (for a given value of \(n\)):

\[
\forall \gamma \in \Gamma^k \quad (|\gamma| = \xi), \forall q \in Q
\]

there exists a set \(U_{q, \gamma} \subseteq Q' \times \Gamma^*\) such that

\[
\forall \xi \in \Gamma^*, \ (p^i, \alpha \xi) \in U_{q, \gamma} \implies (p^i, \beta) - U_{q, \gamma} \cdot 1
\]

**Note:** If the size of the uncertainty zones is bounded, the set is finite.

The smallest set \(U_{q, \gamma}\) having the property is called the uncertainty zone associated with the truncated configuration \((q, \gamma)\). When no ambiguity is possible, \(U_{q, \gamma}\) will just be called an uncertainty zone.

\footnote{More exactly: whatever the value of \(\xi\) such that \((q, \gamma'\xi)\) is a reachable configuration, i.e. \(\exists x, x : (q_o, x) \xrightarrow{\delta} (q, y'\xi)\). We will assume implicitly this restriction.}
If \( G \) is \( E_n(\mathcal{L}) \), the reconstruction of a set which contains \( U_{q,\gamma}\xi \) is easy.

V.3.3 Discussion

In general, programming languages have no \( E_n(\mathcal{L}) \) generator, but the notion of an uncertainty zone associated with a truncated configuration is a natural and convenient tool. In order to use this tool, two strategies are possible:

a) Use truncated configurations instead of the full configurations. This just means that errors "deep in the stack" won't be corrected. More corrections may be performed somewhere else, instead.

b) Make heuristic assumptions about the nature of errors. These assumptions lead to a new transition function \( \delta' \) of \( G \). If they are well chosen, the generator may become \( E_n(\mathcal{L}) \). Examples of possible assumptions are:

- "brackets" missing or in excess are corrected only where no other correction is possible,
- "keywords" when encountered in a correct context are never corrected into an identifier.

These strategies will be studied later. First, we discuss an algorithm which computes the uncertainty zones associated with truncated configurations.
V.3.4 Computation of the uncertainty zones for one symbol look-ahead

The algorithm to compute the uncertainty zones (associated with truncated configurations) follows the same idea as the one for regular languages (see IV.2). It is described in detail in the appendix. Here, we shall just summarize some important characteristics of the algorithm.

We develop the algorithm for $E_n(\ell)$ generators and then we show how to use it in more general conditions. More precisely, for any generator (not necessarily $E_n(\ell)$) such that

$$\forall p, \beta, \quad (p, \beta) \in \mathcal{U}_{q, a} \Rightarrow |a| - \ell \leq |\beta|$$

approximations $W_{q, a'}$ to the uncertainty zones associated with the truncated configurations $(q, a')$ can be computed and used in place of the uncertainty zones. Using these approximated zones is approximatively equivalent to correcting parts of the program which correspond to the inner levels of nesting with respect to the level at which the error is detected. This is quite a natural assumption.

Henceforth, we shall deal only with truncated configurations.

The algorithm terminates if and only if the (proper) uncertainty zones are finite.

Both above properties are true for "practical" programming languages as we shall see now.
V.4  Error-correction of "practical" programming languages

In order to be able to say more about the error-correction process, we want to reduce the class of languages considered. Indeed, not all context-free languages are reasonable programming languages. On the other hand, it is impossible to define exactly what is a "practical" language. However, there is one characteristic, important to error correction, which we believe is common to most programming languages. This characteristic is their bracketed structure. We will define this notion precisely, then study its consequences on the error-correction process.

Bracketed languages have finite uncertainty zones and simple assumptions on the correction of brackets help getting a bounded backward move.

V.4.1 Bracketed context-free languages

Informally, brackets are symbols which mark the beginning and the end of a level of nesting.

Definition. A context-free language which can be represented by a system of equations of the form

\[ A_0 = r_0 \]
\[ A_1 = [1 r_1]_1 \]
\[ \vdots \]
\[ A_k = [k r_k]_k \]

\[ ^1 \text{Our definition of bracketed context-free language is quite different from Ginsburg's one [Gin., 67].} \]
where $\Sigma' = \Sigma - \{ \ldots, l_i, \ldots, l_j, \ldots \}$ and $r_i \subseteq (\Sigma' \cup \Lambda_1 \cup \ldots \cup \Lambda_k)^*$
for $i = 0, \ldots, k$, (the $\Lambda_i$ and $l_j$ need not be different) is
said to be bracketed context-free.

A pair $'[i' \& ']'_i'$ is said to be a pair of left and right
brackets $\ast$

Brackets are thus used to uniquely mark the beginning
and the end of each expression (level of nesting). Let us
give an example of bracketed language. Consider the follow-
ing grammar which defines a subset of Algol's expressions:

\[
<expression> \rightarrow <simple expression> \mid \\
\quad \quad \quad <if\ clause><simple expression>else<expression>
\]

\[
<if\ clause> \rightarrow if<expression>\ then
\]

\[
<simple expression> \rightarrow \textbf{id e n t i f i e r} \mid \\
\quad \quad \quad <expression> ) \mid \\
\quad \quad \quad <function\ d e s i g n a t o r> ; \\
\quad \quad \quad <simple expression> * <simple expression>
\]

\[
<function\ d e s i g n a t o r> \rightarrow \textbf{id e n t i f i e r} \ ( <p a r m . \ list> )
\]

\[
<p a r m . \ list> \rightarrow <expression> \mid <p a r m . \ list>, <expression>
\]
This language can be represented by the following system of equations (represented by transition diagrams):\(^1\):

\[
E : \quad \begin{array}{c}
\text{ident.} \\
A
\end{array} \quad \text{expression}
\]

\[
IC : \quad \begin{array}{c}
\text{ident.} \\
A
\end{array} \quad \begin{array}{c}
\ast \\
\ast
\end{array} \quad \begin{array}{c}
\ast \\
PE
\end{array}
\]

\[
PE : \quad \begin{array}{c}
( \\
E \\
)
\end{array} \quad \text{parenthesized expression}
\]

\[
A : \quad \begin{array}{c}
( \\
E \\
)
\end{array} \quad \text{arguments}
\]

\[
IC : \quad \begin{array}{c}
\text{if} \\
E \\
\text{then}
\end{array} \quad \text{if clause}
\]

\(^1\): \(-\quad-\quad-\) denotes a transition over a variable, \(\ast\) denotes a final state.
The language is bracketed context-free. It can be seen by substituting the transition diagram of 'E' for the transition '— — E — —' in 'PE', 'A', and 'IC'. The brackets are '(' and ')', and 'if' and 'then'.

In Fortran, there is only one pair of brackets: '(' and ')'. In Algol, there are four such pairs: \texttt{begin and end, '() and '}, ', [' and ']', and \texttt{if and then}. In PL/1, there are three pairs of brackets: \texttt{begin and end, do and end, and '()and')}. The brackets in a language are not quite well defined, they may depend on the system of equations chosen to represent the language. For example, in the previous example (hence in Algol), there is an equivalent system which has \texttt{if, else} as a pair of brackets instead of \texttt{if, then}. This fact has no consequences on the subsequent results.

As said earlier, we believe that almost all programming languages are bracketed context-free (we do not know of any counter-example).

\textbf{V.4.2 Finite uncertainty zones and bracketed RCF languages}

When computing the uncertainty zones associated with truncated configurations, we encountered the problem of knowing whether the (proper) uncertainty zones are finite. Uncertainty zones are defined for a given parser, therefore general proofs about them are difficult if not impossible.
We shall prove that the parser consisting of recursive finite automata corresponding to a bracketed RCF language has finite uncertainty zones. Under certain conditions, this result can be generalized to other parsers.

Theorem. Given a RCF language defined by a system of equations of the form:

\[
\begin{align*}
\Lambda_0 &= \tau_0 \bot \\
\Lambda_1 &= \{i, r_1\}_I \\
\vdots \\
\Lambda_k &= \{k, r_k\}_k
\end{align*}
\]

where \( \Sigma' = \Sigma - \{\ldots, i, \ldots, j, \ldots\} \) and \( r_i = (\Sigma' \cup A_1 \cup \ldots \cup A_k)^* \), the corresponding parser, consisting of recursive finite automata, has finite uncertainty zones.

Proof. The proof is immediate: each left bracket, when encountered, results in one symbol being stacked; each right bracket results in one symbol being unstacked. Otherwise, the stack is never changed. Thus, a prefix \( x \) puts the recognizer in state \((p, a)\) where

\[|a| = h = \text{number of left brackets} - \text{number of right brackets in } x\]

and any interpretation \( y \) of \( x \) (with at most \( n \) corrections) puts the recognizer in state \((p', a')\) where

\[|a'| = \text{number of left brackets} - \text{number of right brackets in } y.\]

That is \( h - n \leq |a'| \leq h + n.\)

Therefore, there can be only a finite number of configurations in any \( U_{p, a}. \) Q.E.D.
Clearly, any parser which is such that a bounded number of symbols can be stacked or unstacked after a bracket is encountered and before the next bracket is encountered, will have finite uncertainty zones (they may be however unbounded). This is true for most practical parsers (an example is given in the next chapter).

V.4.3 Bounded backward moves

We shall show that, even for "practical" languages, the backward move may be unbounded. This is generally unacceptable. Therefore, some more heuristic assumptions will be introduced. But, first, we show that it is decidable whether the backward move is bounded.

Theorem. It is decidable whether the backward move (using uncertainty zones associated with truncated configurations or approximations to these zones) is bounded.

Proof. The algorithm for the backward move is described in III.4.2. Clearly, the backward move can be unbounded if and only if constructing $T_i$ going backwards on some string $\ldots a_1a_{i-1}\ldots a_k$, it happens that $T_i = T_j$ for some $i \neq j$.

Let $N$ be the number of truncated configurations and $M$ be the size of the largest (truncated) uncertainty zones. $N$ and $M$ are finite.

There are at most $N \times 2^M$ distinct sets $W$. Hence, by checking the backward move on at most all strings of length $N \times 2^M$. 
either the backward move will stop on each one or the above condition will be satisfied in at least one case. (Note that $N \times 2^M$ is just an upper bound on the length and not the actual length which is far less).

In general, for "practical" languages, the backward move is unbounded. The main reason is that a left bracket missing (or in excess) is likely not to be detected until the point where the right bracket is (or should be) and the distance in-between may be unbounded. Furthermore in most programming languages, brackets can be inserted almost everywhere in a prefix. For example, given the prefix 'u' of an Algol or PL/1 program, 'begin u' is also a correct prefix. This means that in Algol or PL/1 the backward move would always go all the way back. We now take a closer look at this problem.

Brackets which create trouble are brackets 'h' such that if 'ub' is a correct prefix, then 'ub^k' is also correct (possibly within certain limits for k). We shall say that these brackets are repeatable. For example, in Algol all brackets but then are repeatable.

When a repeatable left bracket is missing (or in excess), there are often at least two equivalent corrections for such an error: add (suppress) the left bracket or suppress (add) a right bracket. For example, unbalanced parenthesis can always be corrected by adding or deleting right parenthesis where the error is detected. However, even for conventional languages, it is not always the case. Consider the following incorrect
Algol program:

begin integer i; if i = i then i := i; i := i; i := i end
else i := i end

The only unique action which corrects the string is to insert a begin after the then.

As suggested by the above discussion, in an attempt to bound the backward move, one can make the following heuristic restrictions on the handling of repeatable brackets by the error-correction process:

1) Such brackets, whether missing or in excess will be corrected only by adding or subtracting brackets after the point where the error is detected.

2) Insertion or deletion of brackets is done only if nothing else is possible (in order to be able to find equivalent prefixes).

The restrictions correspond to a change in the $\delta'$ function (see III.4.1). Uncertainty zones are computed using this new function.\(^1\)

Here we depart from our formal approach of trying to

---

\(^1\)We call these restrictions "heuristic," because they lead to an error-correction process which is tractable, but not always as good as the one defined originally (i.e. more corrections may be needed).

\(^2\)One has to be careful, however, when computing these zones, not to create brackets by the following operations: substitute a bracket for a symbol, then insert this symbol after the bracket.
correct errors in the best possible way. These assumptions lead to a "good" error-correction process, but, since the approach is pragmatic, there is no formal result to support this claim. Only experiments can do it.

The above restrictions are generally, but not always, sufficient to bound the backward move. If this move is not bounded one can either accept it (if it happens in rare cases) or make further assumptions. Some of these are discussed in the next chapter. The choice of the additional assumptions should be based on an analysis of which corrections require such an unbounded move.

V.5 Summary and description of the process

For practical purposes, a heuristic handling of brackets is needed. With these restrictions the new generator $G$ may or may not be $L_2^*(z)$ (see example in next chapter). In any case, uncertainty zones or approximations to these zones can be computed in a convenient form (truncated configurations).

The error-correction process is similar to the one for regular languages.

The left-to-right syntax check is done using a parsing technique which detects incorrect prefixes (preferably recursive finite automata, as will be justified later).

Beacons, if they exist, can be used for the backward move. Otherwise, the problem is which information to store
during the forward move in order to be able to go backward using uncertainty zones. Fortunately, it is not necessary to store the whole configuration at each step. Storing the state of the finite control and information about the stack movement each time there is ambiguity about what might precede is sufficient.

As for the forward move, the only problem is the handling of repeatable brackets. In order to avoid having prefixes being nonequivalent only because of differences in bracketing, brackets are inserted or deleted only if nothing else is possible. More precisely:

- if the next symbol is illegal, a bracket may be inserted (by illegal, it is meant that if \((p^i,a)\) is the current state, 'a' the next input symbol, there is no \((q,\beta)\) such that \(a : (p,a) \vdash \delta \rightarrow (q,\beta)\)),
- if a bracket has just been read and the next symbol is illegal, the bracket may be deleted,
- if a bracket appears at a place where it is illegal, it may be deleted,
- a symbol can either be replaced by a bracket or suppressed, but not both.
CHAPTER VI: SELECTING AN INTERPRETATION, MISCELLANEOUS PRACTICAL SUGGESTIONS, AND EXAMPLES

VI.1 Selecting an interpretation

When the interpretations of a prefix are equivalent, which one is selected is immaterial from a purely syntactical point of view. Of course, for the user, the quality of the error-correction process is closely related to the quality of these selections.

Which interpretation is selected is not a language-theoretical problem. It is linked to the actual representation of the symbols (a several letter identifier is likely not to be the mispelling of ';') and to the semantic of the language. It relies heavily on statistical considerations which data can be estimated using a representative sampling or acquired via some learning process.

A detailed study of this selection would be outside the scope of this dissertation. We will only examine some possible approaches to this problem.

A first approach is to try to minimize the total number of corrections. This is the approach taken in the second chapter, but, the problem of selecting a string among those with a minimal number of corrections remains. A possible solution is to assign a probability to each error (hence to
each correction) and to select the string with the maximal probability of being the "good" interpretation. In general, given the corrections $c_1, c_2, \ldots, c_k$, there will be an objective function $f(c_1, \ldots, c_k)$ which is to be maximized over all potential interpretations with a minimum number of corrections.

Now, there is no reason for trying only to minimize the function over strings with a minimum number of errors. One may decide that it is better to choose a string with two likely errors rather than a string with a very unlikely one. In this case, the objective function will be minimized over all potential interpretations with at most $n$ corrections (a bound is still needed if one wants a left-to-right process).

Errors need not be independent from one another. For example, inversion of two symbols may be more likely than the two basic errors (addition plus deletion) we consider. But, if one decides to consider errors as independent from one another, $f$ might be expressed in the following form:

$$f(c_1, \ldots, c_k) = f(c_1) \ast \ldots \ast f(c_k)$$

where $\ast$ is some associative binary operator. Such a function can be evaluated as the interpretations are constructed. Whenever two interpretations $u$ and $v$ are equivalent, the one with the minimal objective function, say $v$, is discarded and the construction of the interpretations continues with $u$ and the other interpretations.
f(c), for some correction 'c', may be evaluated using semantic information and information about the actual representation of the symbol or symbols involved. In particular, if 'c' consists in replacing a symbol (a) by another (b), the value of f should take into account the probability of b being misspelled as a. The techniques for spelling correction mentioned in 1.3 could be very fruitfully used for such a purpose.

VI.2 Miscellaneous practical suggestions

1) G need not be actually implemented in the process. It can be easily simulated using M and making guesses about the input symbols to M.

The subsequent suggestions are heuristics which can be used to simplify the process, but may cause the loss of possible corrections.

2) Whenever the probability of some error is negligible, the transition in G corresponding to it can be eliminated. This leads to smaller uncertainty zones, increasing the likelihood of having a bounded backward move and of having beacons. The forward move is also simplified since the number of states G can be in at a given state is reduced. An example concerning beacons is given later.

3) During the forward move, it may happen that no interpretation exists for the prefix. For example, one might be
scanning \( xa \ldots \) with two nonequivalent interpretations, \( u \) and \( v \), of \( x \) and no interpretation of \( xa \). In this case, one might select either \( u \) or \( v \) to replace \( x \) and go back to the standard left-to-right scan anyway. Alternatively, one can use a more primitive \textit{recovery} technique in order to discover more errors (for example McKeeman's technique which is mentioned in I.3).

4) The forward move may be unbounded. Indeed, examine the following incorrect Algol statement:

\[
...; i = \text{<expression>} \text{ then } ...
\]

An error is detected at \( '=' \). There are at least two interpretations of \( '...; i = \text{<expression>}' \) which are

\( '...; i := \text{<expression>}' \) and \( '...; if i = \text{<expression>}' \).

The first one is discarded when \textit{then} is read. Since the size of \( '\text{<expression>}' \) is unbounded, the forward move is unbounded too.

This may be undesirable. In which case, one may set some limit \( \delta_f \) on the forward move. If all interpretations are not equivalent \( \delta_f \) symbols after the point where the error is detected, one interpretation is selected anyway.

Similarly, one may set an arbitrary limit, \( \delta_b \), on the backward move. Clearly, in such cases one cannot prove anything on the "goodness" of the process.

Let us note the following.
Theorem. Given a parser, for a language $L$ such that at each transition at most $m$ symbols of the stack are examined, if backward and forward move are arbitrarily bounded then $C_n(L)$, the set of all corrected string, is deterministic context-free.

Proof. Let $l_b$ and $l_f$ be the bounds on backward and forward move respectively. Over a string of length $(l_b + l_f)$ the generator $G$ can examine at most $(l_b + l_f + n)m$ symbols of the stack (n is due to the fact that $n$ symbols might be inserted).

A finite table is constructed which arguments are pairs of a truncated configuration and a string of input symbols

$$((p, a), uv) \quad \text{where } |a| = (l_b + l_f + n)m$$

$$|u| = l_b$$

$$|v| = l_f$$

such that, for all $k$, the recognizer started in state $(p, a_k)$ and running over $uv$ detects an error at the first symbol of $v$. The behavior of the error-correction process over $uv$ can be completely simulated knowing only $(p, a)$ and $uv$. The string $uv$ is replaced by $w$. Thus, the table associate to each argument $((p, a), uv)$ the value $w$.

Now, the deterministic recognizer for $C_n(L)$ works as follows:

- the finite control contains the above (finite) table,
- a register to contain $l_b + l_f$ input symbols, and a copy
of the finite control of $M$.
- at each step, the left-most symbol of the register is used to determine the new simulated state of $M$.
The register is shifted left and a new symbol entered in its right-most cell. If the state $p$ of the copy of the finite control of $M$, the $(t_b+t_f+n)_m$ top-most symbols of the stack, and the content $uv$ of the register appear as argument in the table, then the content of the register is replaced by the corresponding $w$.

This deterministic PDA simulates the error-correction process.

5) One could use two values of $n$, the first one, $n_b$, used to compute the backward move, the second one, $n_f$, used during the forward move. A small $n_b$ allows for beacons, while $n_f$ can be chosen such as to provide for a sufficient number of corrections. We will use this technique in our further examples.

VI.3 Examples

We will study the error-correction process on the same subset of Algol for two parsing techniques: a top-down technique, recursive finite automata, and a bottom-up one, SLR(1) parse.

The language studied is block structured and has declarations, assignment statements, and conditional statements. $n_b$ is $1$ ($n_b$ is the value of $n$ used to compute the
uncertainty zones, \( n_f \) the value used for the forward move).

In order to reduce the size of the uncertainty zones, we will make the natural assumption that "keywords"

\[
\text{begin, type, if, then, else, end}
\]

when they appear in a string are either correct or in excess; they do not represent the misspelling of some identifier. However, an identifier may be the misspelling of some keyword.

This restriction is made in order to have a compact example. We also performed the computation without these assumptions for recursive finite automata. The recognizer was still

\[ E_1(1) \]

and only two beacons disappeared, with respect to the following example. They are

\[((16,10),id)\text{ and }((11,-),id)\].

**VI.3.1 Using recursive finite automata**

The language is of the form

\[
L = B \downarrow
\]

\[
B = \text{begin } f(B,E) \text{ end}
\]

\[
E = g(E)
\]

where the automata are:
The language is index bracketed. To show it substitute

for $E$ in $B$, where $E'$

$$
E' \leftarrow ( \text{id} )
$$
The first form is just more natural. The brackets are: \texttt{begin, end, '(' , and ')'} . All brackets are repeatable.

The recognizer is $E_1(1)$. Its uncertainty zones are computed using the algorithm described in the appendix, which yields:

(note: whenever $WZ$, $(p,aZ)\in_{-} U_q, Z$, we denote it by $(p,a)\in_{-} U_q, -$

\begin{align*}
U_{1, Z_0} &= \{(1, Z_0)\}
U_{2, -} &= \{(2, -) (4', -) (3', -)\}
U_{3, -} &= \{(3, -) (2', -) (4', -)\}
U_{4, -} &= \{(4, -) (5', -)\}
U_{5, -} &= \{(5, -) (2', -) (2', 7) (3', -) (4', -) (8', -)
\quad (9', -) (9', 7) (19', 7) (19', 10) (19', 14)\}
U_{6, -} &= \{(6, -)\}
\end{align*}

7 is a state from which the automaton returns, performing an $\epsilon$-move. The machine is never in state 7, after a completed transition.

\begin{align*}
U_{8, -} &= \{(8, -) (15', 7) (15', 14) (16', 7) (16', 14)
\quad (19', 7)\}
U_{9, -} &= \{(9, -) (2', -) (8', -) (11', -)\}
\end{align*}

No transition terminates in 10

\begin{align*}
U_{10, -} &= \{(11, -) (19', 10)\}
U_{12, -} &= \{(12, -) (9', -) (11', -)\}
\end{align*}
\[ U_{13,-} = \{(13,-)\} \]

No transition terminates in 14

\[ U_{15,-} = \{(15,-)\} \]

\[ U_{16,7} = \{(16,7) (8',c) (19',7)\} \]

\[ U_{16,10} = \{(16,10) (19',10)\} \]

\[ U_{16,14} = \{(16,14) (8',c) (19',14)\} \]

\[ U_{16,18} = \{(16,18) (19',c) (19',18)\} \]

\[ U_{17,-} = \{(17,-)\} \]

No transition terminates in 18

\[ U_{19,7} = \{(19,7) (6',c) (16',7) (17',7) (5',c)\} \]

\[ U_{19,10} = \{(19,10) (5',c) (12',c) (16',10) (17',10)\} \]

\[ U_{19,14} = \{(19,14) (13',c) (16',14) (17',14)\} \]

\[ U_{19,18} = \{(19,18) (16',18) (17',c) (17',18)\} \]

All but the six following transitions have such a property:

( (2,-) , id)

( (3,-) , id)

( (8,-) , id)

( (9,-) , id)

( (16,7) , id)

( (16,14) , id)

This means that the backward move is at most two symbols.

The correction of the string:

\begin{verbatim}
begin id ; id +( id ; ) id + id ) ; type id ; id + id 
then id +( id ; else id ( id + end )
\end{verbatim}
is illustrated by the following figure (we chose \( n_x = 2 \) for the forward move).

The figure just shows the substring selected to replace each cluster of errors. Braces show the extent of the backward and forward moves and the substring selected. Vertical arrows point to symbols at which errors are detected. A detailed description of the process would be very similar to the one in chapter IV.

\[
\begin{align*}
\text{begin} & \quad \text{id} ; \quad \text{id} + ( \quad \text{id} ; \quad ) \quad \text{id} + \\
\text{begin} & \quad \text{type} \quad \text{id} ; \quad \text{id} + ( \quad \text{id} ) ; \quad \text{id} + \\
\text{id} & \quad ) ; \quad \text{type} \quad \text{id} ; \quad \text{id} + \quad \text{id} \quad \text{then} \quad \text{id} \quad + ( \\
\text{id} & \quad ) ; \quad \text{begin} \quad \text{type} \quad \text{id} ; \\
\text{id} & \quad ; \quad \text{if} \quad \text{id} + \quad \text{id} \quad \text{then} \quad \text{id} \\
\text{id} & \quad ; \quad \text{else} \quad \text{id} \quad ( \quad \text{id} + \quad \text{end} \quad \text{)} \\
\text{id} & \quad ) \quad \text{else} \quad \text{id} + \quad \text{id} \quad \text{end} \quad \text{end} \quad \text{)} \\
\text{id} & \quad ; \quad \text{id} + ( ( ( \ldots ( \ldots 
\end{align*}
\]
among its interpretations, for \( n = 2 \), appear:

\[
... \text{ if } (((\ldots)( \ldots)
\]

and vice-versa. Also

\[
... \text{ then id } + (((\ldots)( \ldots)
\]

and

\[
... \text{ then if } (((\ldots)( \ldots)
\]

Approximations to the uncertainty zones can, however, be computed.

VI.3.2 Using an SLR(1) parser

A grammar for the language described in the previous section is:

\[
\begin{align*}
(0) & \quad P \rightarrow B \downarrow & \text{(P: program)} \\
(1) & \quad B \rightarrow BB \text{ end} & \text{(B: block)} \\
(2) & \quad BB \rightarrow BH S & \text{(BB: block body)} \\
(3) & \quad BB \rightarrow BB \text{ ; } S & \\
(4) & \quad BH \rightarrow \text{begin} & \text{(BH: block head)} \\
(5) & \quad BH \rightarrow BH \text{ D ;} & \\
(6) & \quad D \rightarrow \text{type id} & \text{(D: declaration)} \\
(7) & \quad D \rightarrow D , \text{ id} & \\
(8) & \quad S \rightarrow IS & \text{(S: statement)} \\
(9) & \quad S \rightarrow SS & \\
(10) & \quad IS \rightarrow IL SS & \text{(IS: if-statement)} \\
(11) & \quad IL \rightarrow IL IT & \text{(IL: if-clause list)} \\
(12) & \quad IL \rightarrow IT SS \text{ else} &
\end{align*}
\]
(13) IL \rightarrow IT \ SS
(14) IT \rightarrow if \ E \ then \hspace{1cm} (IT: \ if-then \ clause)
(15) SS \rightarrow B \hspace{1cm} (SS: \ simple \ statement)
(16) SS \rightarrow id \ E
(17) E \rightarrow E + T \hspace{1cm} (E: \ expression)
(18) E \rightarrow T
(19) T \rightarrow ( \ E ) \hspace{1cm} (T: \ term)
(20) T \rightarrow id

This grammar is SLR(1) (SLR(k) grammars and parsers were introduced by DeRemer [DeR., 71]). The Characteristic Finite State Machine (finite control of the one-symbol look-ahead parser) is the following one:
The parsing algorithm is the following.

Begin with the CFSM in state 0 and 0 in the stack.
The string to be parsed is \( n \).

1) Let \( a = n \).

2) Starting the CFSM in whatever state it was in prior to this step, cause the CFSM to begin reading \( a \) and to store on the stack the name of each state entered.

3) When the CFSM enters a "reduce" state ('\( \sqcup \) ' in the previous diagram), set \( a \) to the suffix of \( a \) not yet read. Let the production associated with the "reduce" state be \( A \rightarrow \omega \) \( (0 \xrightarrow{\#i} \sqcup) \) means that the \( i \)th production is associated with the "reduce" state). Pop the top \( |\omega| \) items of the stack. If \( A = P \) the parse is complete, so stop; otherwise return the CFSM to the state whose name is on the top of the stack, set \( a = A\alpha \) and go to step (2).

Note: whenever there is ambiguity about the next state, one symbol look-ahead is used. For example in state 19 the "reduce" state is entered if the next symbol is not else. For more details, see DeRemer's paper.

Each state (and stack symbol) corresponds to one terminal or nonterminal. The stack corresponds to the prefix of a sentential form obtained by left-most reduction of \( n \). Thus, upon examination of the grammar or of the CFSM, one sees that:

1) in one transition, at most seven (top-most) stack
symbols are examined. This seven symbols are examined when there is a transition from state
(33, 29 23 25 17 13 7 2 ... )
over ';' or end, i.e. when a reduction of the sentential forms ... BB ; IL id (E) ; ... or
... BB ; IL id (E) end ... is performed,
2) between two brackets one can stack only a bounded number of symbols; the size of the uncertainty zones
is, therefore, bounded.

For a stack symbol to appear above another in the stack, implies that there is a transition in the CFSM from the
lower symbol to the higher. The function $\phi_1$ is therefore easy to compute.¹ For example,

$\phi_1(q_3(c^3)) = \{0, 3, 7, 13\}$.

The first remark leads us to test if the parser is $E_1(7)$, which it is. The uncertainty zones are:

(Note: if $\forall \xi, (p, \alpha \xi) \in U_{q, \beta}$, we only list $U_{q, \beta} = (p, \alpha)$; see appendix)

$U_{0,0} = \{(0,0)\}$
$U_{4,-} = \{(4,-)\}$
$U_{5,10} = \{(5,10)\}$
$U_{6,2} = \{(6,2)\}$

¹$\phi$ is defined in the appendix.
\[ U_{7,2} = \{(7,2) (6',2,3) (6',2,7,2) (24',25,17,3)\}
\[ (24',25,17,13,3) (24',25,17,13,7,2) (6',13,2)\]
\[ (24',25,17,2) (28',30,25,17,3) (28',20,25,17,7,2)\]
\[ (28',30,25,17,13,3) (28',30,25,17,13,7,2)\}\n
\[ U_{15,3} = \{(15,3) (4',e) (18',9,3)\}\]
\[ U_{15,7,2} = \{(15,7,2) (7',2)\}\]
\[ U_{15,13} = \{(15,13) (26',19,13) (27',21,15) (27',21,15,13)\}\]
\[ U_{17,3} = \{(17,3) (18',9,3) (34',3) (15',3) (24',15,3)\}
\[ (4',e) (37',35,9,3) (36',34,6) (24',15,3)\}\n\[ U_{17,13} = \{(17,13) (4',13) (15',13) (27',21,15)\}
\[ (27',21,15,13) (26',19,13)\}\n
\[ U_{17,7,2} = \{(17,7,2) (15',7,2) (7',2) (24',15,7,2)\}
\[ (24',7,2) (24',28,30,25,17,3) (24',28,30,25,17,7,2)\]
\[ (24',28,30,25,17,13,3) (24',28,30,25,17,13,7,2)\]\n
\[ U_{18,9,3} = \{(18,9,3) (36',34,3) (35',9,3) (37',35,9,3)\}\n\[ U_{23,-} = \{(23,-)\}\]
\[ U_{24,15} = \{(24,15) (23',15) (15',e) (17',e)\}\]
\[ U_{24,23} = \{(24,23) (23',23) (23',e)\}\]
\[ U_{24,25} = \{(24,25) (25',e) (23',25)\}\]
\[ U_{24,28} = \{(24,28) (28',e) (23',28)\}\]
\[ U_{25,17} = \{(25,17)\}\]
\[ U_{26,19} = \{(26,19) (24',25,17) (6',2,13)\}\]
\[ U_{27,21} = \{(27,21) (24',e) (24',28,21) (33',29,23)\}\]
\[ U_{28,21} = \{(28,21) (24',e) (24',28,21) (33',29,23)\}\]
\[ U_{28,29} = \{(28,29) (24',e) (33',29) (33',29,23)\}\]
\[ U_{28,30}^{25} 17 \ 3 = \{ (28,30 \ 25 \ 17 \ 3) (7',2) (24',28 \ 30 \ 25 \ 17 \ 3) (24',25 \ 17 \ 3) \} \]

\[ U_{28,30}^{25} 17 \ 7 \ 2 = \{ (28,30 \ 25 \ 17 \ 7 \ 2) (7',2) (24',28 \ 30 \ 25 \ 17 \ 7 \ 2) (24',25 \ 17 \ 7 \ 2) \} \]

\[ U_{28,30}^{25} 17 \ 13 \ 3 = \{ (28,30 \ 25 \ 17 \ 13 \ 3) (7',2) (24',28 \ 30 \ 25 \ 17 \ 13 \ 3) (24',25 \ 17 \ 13 \ 3) \} \]

\[ U_{28,30}^{25} 17 \ 13 \ 7 \ 2 = \{ (28,30 \ 25 \ 17 \ 13 \ 7 \ 2) (7',2) (24',28 \ 30 \ 25 \ 17 \ 13 \ 7 \ 2) (24',25 \ 17 \ 13 \ 7 \ 2) \} \]

\[ U_{33,29} = \{ (33,29) \} \]

\[ U_{34,3} = \{ (34,3) (4',c) (18',9 \ 3) \} \]

\[ U_{35,9} = \{ (35,9) (18',9) (36',34) \} \]

\[ U_{36,34} = \{ (36,34) (17',c) \} \]

\[ U_{37,35} 9 = \{ (37,35 \ 9) (17',c) (35',9) \} \]

All but the following transitions are beacons:

(7,2) , id
(15, -) , id
(18,9 3 , id
(28,30 25 17) , id
(34,3) , id
(35,9) , id

Here also, the backward move is at most two symbols.
The number of elements in the table is much greater than for the recursive finite automata. This is because two strings which lead to the same state of the recursive finite automata may lead to two different states of this SLR(1) parser. The latter memorizes more information in a state than strictly needed. For example,

'begin id id + ( ' and 'begin id ( '
lead to the same state, \((17,7 Z_o)\), of the first parser and to two states, \((23,28 30 25 17 3 0)\) and \((23,25 17 3 0)\), of the SLR(1) parser.

This also indicates that the forward move may be larger for this parser. Indeed, the string studied in the previous section is corrected as follows:

\[
\begin{align*}
\text{begin id ; id + ( id ; ) id + id } & ; \\
\text{begin id ; id + ( id ) } & ; \\
\text{id - id } & ; \\
\text{id ; begin } & \\
\end{align*}
\]

\[
\begin{align*}
\text{type id ; id + id then id + } & ; \\
\text{type id } & ; \\
\text{if id + id then id } & \\
\end{align*}
\]
VI.3.3 Discussion

Recursive finite automata yield a simpler error-correction process. Tables are smaller and involve the examination of less stack symbols. Note, however, that the grammar of the language parsed can be transformed to yield an SLR(1) parser for which less symbols of the stack are examined. Gries [Gri., 68] has described such a technique. It mainly consists in encoding more information in stack symbols, thus using less symbols to keep the information on the prefix already scanned. In any case, recursive finite automata give a more "compact" process.

VI.4 The error-correction process as part of a compiler

Most parsers are based on a grammar of the language to be analyzed. Grammars are designed not only to describe correct programs but also to facilitate translation: they convey more information than strictly needed for syntax checking. For example, grammars for arithmetic expressions generally indicate precedence relationships between operators, which is not needed for syntax checking. Compare:
\[ \begin{align*} 
E + E & \quad + \quad T \quad | \quad T \\
T + T & \quad * \quad F \quad | \quad F \\
F + F & \quad \uparrow \quad P \quad | \quad P \\
P + \text{var} & \quad | \quad (\ E \ ) \\
\end{align*} \]

with
\[ \begin{align*} 
E + E & \quad + \quad P \quad | \quad E + P \quad | \quad E \uparrow P \\
P + \text{var} & \quad | \quad (\ E \ ) \\
\end{align*} \]

and with
\[ \begin{align*} 
(\ 
\text{var} \\
\uparrow, +, \uparrow \\
\text{E} \\
) \\
\end{align*} \]

Wilcox [Wil., 71] affirms that most practical compilers for complex programming languages have distinct passes for syntax analysis and for translation. Thus, the grammar used during syntax analysis can be as simple as possible. Recursive finite automata are a fast and compact way to perform this analysis.

Having a distinct pass for syntax analysis is also better when error-correction is performed for the following reason. During the pass which includes syntax analysis, some work is done in parallel with this analysis. When a backward move takes place, part of this work must be undone which may be difficult. Beacons, when they exist in sufficient number, are very useful in this respect: each time
a beacon is encountered, one knows that everything before it is permanent. One might perform syntax analysis by bursts: going from one beacon to another and, then, performing all the work relative to the substring between beacons. This may be difficult to implement. If syntax analysis is done in a distinct pass, the only work done concurrently is, in general, to create entries in the symbol table. One can tag these new entries by "provisional." When a beacon is encountered all "provisional" entries are made "permanent." When an error is detected, all "provisional" entries are deleted and one goes back to the last beacon. Admittedly, this method is just outlined.
CHAPTER VII: CONCLUSION

VII.1 Summary of the dissertation

In this thesis, we have developed a model for error correction. Treatment of syntax errors in the literature is heuristic. Most often, it is recovery rather than correction that is undertaken.

Our approach has been to study formally and define the notion of error. Then, using this definition, we have modeled a systematic error-correction process. This process makes local corrections over cluster of errors, using the context around the errors to determine the corrections and to insure that the different local corrections performed on the string do not interfere with one another. The error-correction process can be naturally embedded in many left-to-right syntax checking processes. It uses the recognizer both to detect errors and to find possible corrections.

The process has two modes: a "standard mode" used for syntax checking, and an "error-correction mode" used for determining the context of a cluster of errors and for finding all possible corrections of these errors. In the "standard mode," the syntax is checked at the same speed as if no error-correction mechanism is implemented. Thus, for strings which contain no errors, no price is paid for
the presence of this mechanism. The "error-correction mode" consists of two phases: the backward move which locates the left context of the cluster, and the forward move which constructs possible corrections and locates the right context of the cluster.

This process seems to us the most natural way to perform left-to-right syntax checking and error correction.

We showed that in an analyzed string there may be points which are outside the range of any backward move. Such points, which correspond to some of the transitions in the recognizer, have been called beacons.

We introduced the notion of bracketed context-free languages to model programming languages. Assuming that brackets are corrected only when no other correction is possible, and that errors in deep levels of nesting are neglected, we showed how our process can be used to correct syntax errors in programming languages.

VII.2 Practicality and suggestion for future work

The process has been described with enough precision to make clear how it can be implemented. A syntax analyser including the error-correction mechanism could even be generated by a metacompiler which would, at metacompile time, compute uncertainty zones and beacons.

However, the process involves a good deal of work. Indeed, our purpose has been to describe, subject to
certain conditions, the "best" error-correction process. As seen in chapter II, to find all interpretations of a substring of a string from a context-free language requires a time which may grow as fast as the cube of the length of the substring. It is, therefore, reasonable to look for just a "good" process. By "a good process," we mean a process which would not make substantially more correction and which requires less work. We feel we have provided a conceptual framework for developing, in a systematic and orderly way, heuristic assumptions and restrictions which lead to such a "good" error-correction process. A few examples of possible assumptions have been given in chapter VI.

We feel that the process is handling brackets in a good and natural way, but it is certainly not the only one possible. Some experimental research is needed to determine which string to select among several equivalent interpretations.

Language designers should be concerned about the complexity of error correction in the languages they are creating. Beacons greatly simplify this process. Thus, the designer should try to have numerous beacons in his language. It would, therefore, be very helpfull to have simple sufficient conditions for a transition to be a beacon.

Correction of syntax errors concerning language features not described by a context-free grammar (e.g., erroneous number of indices in an array reference) and correction of semantic
errors are certainly worth more study than has been done at
the present time. Realistic models for these syntactic
features and for semantic may be needed in order to under-
take this study.
APPENDIX: COMPUTATION OF THE UNCERTAINTY ZONES FOR ONE-SYMBOL LOOK-AHEAD PARSERS

The algorithm is developed for $E_n(\epsilon)$ generators. Then, we show how to use it in more general conditions.

A.1 Transitions from truncated configurations

We look for an algorithm to construct uncertainty zones associated with truncated configurations of one-symbol look-ahead recognizers. We will have to deal with these truncated configurations. Therefore, we now study properties of such recognizers with respect to truncated configurations.

A first problem is, given a truncated configuration $(q, a)$ and the input symbol $a$, can one construct $(p, \beta)$ such that

$$\forall \xi, a : (q, a\xi) \rightarrow (p, \beta \xi)$$

If one is sure that to construct any transition the recognizer never looks down in its stack more than $m$ symbols, where $m < |a|$, then $(p, \beta)$ can be constructed without any difficulty.

Recall that $\rightarrow$ represents a sequence of $\epsilon$-moves followed by a move reading the input symbol. Since any move involves only the examination of a bounded amount of stack symbols, this bound $m$ exists if and only if the parser can make only a bounded number of consecutive $\epsilon$-moves from any
reachable configuration. This property is decidable.

Theorem. It is decidable whether there is a bound on the number of consecutive $\varepsilon$-moves a one-symbol look-ahead parser can perform from any reachable configuration.

Proof. Given the recognizer $M$, transform it into a non-deterministic PDA $M'$ which will:

1) guess fictitious strings of symbols and perform the moves $M$ would do on this string ($M'$ performs $\varepsilon$-moves while guessing),

2) randomly switch to a mode where first $M'$ guesses an input symbol $a$ to $M$, then for each actual input symbol $\#$ (to $M'$) $M'$ simulates one $\varepsilon$-move of $M$ or accepts the string of $\#$'s if no $\varepsilon$-move is possible.

The language accepted by $M'$ is

$$\{\#^i \mid M \text{ can execute } i \text{ consecutive } \varepsilon\text{-moves from some configuration and for some input symbol } a\}.$$

This language is finite and only if the bound exists. Now, it is decidable whether a context-free language is finite [Hop., 69]. Q.E.D.

Fortunately, in practical cases, one is not obliged to use such a clumsy technique.

---

1This proof and the next one have been suggested by T. Szymanski.
For recursive finite automata, the problem is equivalent to determining if there is a possible infinite sequence of returns without being obliged to read an input symbol. A simple examination of the automata yields the answer.

For bottom-up techniques, the content of the stack generally represents the head of the sentential form. An unbounded sequence of ε-moves corresponds then to the existence of right recursive productions in the grammar.

When a bound, say \( b \), on the number of ε-moves is found, one can find a bound \( m \) on the maximum number of stack symbols examined during a transition ('|-') by multiplying \( b+1 \) by the maximum number of stack symbols involved during a move.

Henceforth, we will only consider parsers for which a bound exists and truncated configurations \((q,a)\) where the length of \( a \) is greater than this bound.

Thus, given \((q,a)\) - with \(|a| > m\) - and the input symbol \( a \) we know how to construct \((p,\beta)\) such that

\[
\forall \xi, a : (q,a\xi) \rightarrow (p,\beta\xi) .
\]

We now extend the definition of '|-' to truncated configurations.

**Definition.** \( a : (q,a) \rightarrow (p,\beta) \) \(|a| = |\beta| = \xi \)

if and only if \( \exists \xi, \xi' \) such that \( a : (q,a\xi) \rightarrow (p,\beta\xi') \).

We will want to compute all such \((p,\beta)\).
If $a : (q, a \xi) \rightarrow (p, \beta \xi)$ for all $\xi$ (for which $(q, a \xi)$ is reachable), with $|a| = \xi$ either:

1) $|\beta| \geq \xi$. Then $\beta = \beta' \beta''$ with $|\beta'| = \xi$ and $a : (q, a \xi) \rightarrow (p, \beta')$, or

2) $|\beta| < \xi$, then we have to find the set of all strings $\xi$ such that $\forall \xi, a : (q, a \gamma \xi) \rightarrow (p, \beta \gamma \xi)$ (with $|\beta| + |\gamma| = \xi$).

That is, given $(q, a)$, we have to be able to find all strings of length $\xi - |\beta|$ which may appear below $a$ in the stack. We can do it by induction: given $(q, a)$ find the stack symbols which can appear directly below $a$ then, using this information, find the symbols two levels below $a$ in the stack, and so on.

**Definition.** $\phi_1(q, a) = \{z \mid \exists \xi, x, x : (q_0, z_0) \rightarrow^* (q, a \gamma \xi)\} \circ \phi_1(q, a)$ is the set of all symbols which may appear directly below $a$ on the stack.

We shall show that $\phi_1(q, a)$ is constructible for general one-symbol look-ahead parsers. But, let us first note that $\phi_1(q, a)$ can usually be found very easily by direct observation of the parser. For example, in recursive finite automata $a$ represents a sequence of calls. To find $\phi_1(q, a)$ it is sufficient to find where this sequence can have started. In fact, it depends only of the first call on the sequence. For LR(1)-type parsers, $\phi_1(q, a)$ can be deduced from the finite control (called characteristic finite state machine by DeRemer [DeR., 71]). $\phi_1(q, a)$ is the set of all possible
predecessors of the right-most symbol, of \( a \) in the finite control.

In the general case:

**Theorem.** \( \phi_1(q,a) \) is constructible.

**Proof.** (Given \( (q,a) \), we will examine all \( Z \)'s in \( \Gamma \) (a finite set) and decide whether \( Z \in \phi_1(q,\alpha) \). To do it, given \( M \), we construct the pda \( M' \) which successively:

1) guess a fictitious input string and simulates \( M \) over it,

2) checks that the state of \( M \) is \( q \) and the stack is of the form \( aZ\xi \) for some \( Z \) and \( \xi \). Then accepts \( Z \) if \( Z \) is the unique input symbol.

The language accepted by \( M' \) is \( \phi_1(q,\alpha) \). It is effectively decidable whether a given string is accepted by a pda. Thus, for any \( Z \) it is decidable whether it is in \( \phi_1(q,\alpha) \).

We generalize the definition of \( \phi \) to strings of length \( k \):

**Definition.**

for \( k > 1 \), \( \phi_k(q,\alpha) = \{ \gamma Z \mid \gamma \in \phi_{k-1}(q,\alpha) \text{ and } Z \in \phi_1(q,\alpha \gamma) \} \).

\( \phi_k(q,\alpha) \) is the set of strings of length \( k \) which can appear below \( a \) in the stack.

**Remark:** We have been careless about configurations whose stack is less than \( \xi \) symbols. Without loss of generality, we assume that \( Z_0 \)--the initial stack symbol--is uniquely used to mark the bottom of the stack and that truncated
configurations where the whole stack is represented (because it is less than \( n \) symbols deep) are padded with fictitious \( Z_0 \)'s to get a length of \( n \).

A.2 The algorithm

The algorithm to compute uncertainty zones (associated with truncated configurations) follows the same idea as the one for regular languages (see IV.2).

We shall assume that the depth to which the recognizer looks into its stack to construct a transition is bounded and that \( n \) -- the size of the part of the stack represented in the truncated configurations -- is greater than this bound.

We shall use the following functions:

1) M-transform\((q,a,a)\) which computes the truncated configuration \( (p,\emptyset) \) such that \( \forall \xi, a:(q,a\xi) \vdash \delta' (p,\emptyset \xi) \). If no such configuration exists, M-transform\((q,a,a)\) is undefined.

2) G-transform\((q,a,a)\) which computes the set of transform of \( (q,a) \) by \( \delta'_1 \) over \( a \):

\[
G\text{-transform}(q,a,a) = \{ (p,\emptyset) \mid \forall \xi, a : (q,a\xi) \vdash \delta'_1 (p,\emptyset \xi) \}.
\]

G-transform is extended to map sets of configuration:

\[
G\text{-transform}(T,a) = \{ (p,\emptyset) \mid \exists (q,a) \in T, \forall \xi, a : (q,a\xi) \vdash \delta'_1 (p,\emptyset \xi) \}.
\]

G may need more information from the stack than just the \( n \) top-most symbols \( a \); in this case G-transform\((q,a,a)\) is undefined.
3) \( \phi_k(q,a) \) which has been defined in the previous section.

4) "first" and "rem" which are defined as follows:

\[
\begin{align*}
\text{given a string } & a = a_1 a_2 \ldots a_j , \\
\text{first}(m,a) & = a_1 a_2 \ldots a_m \\
\text{rem}(m,a) & = a_{m+1} \ldots a_j \\
\end{align*}
\]

Note that the definitions of these functions do not use the fact that the recognizer is \( E_n(2) \).

The algorithm to compute the uncertainty zones for \( E_n(2) \) recognizers is now given. The sketch of a proof follows. It provides additional explanations.
[0] initialize $W_{q_0, z_0}$ to $(q_0, z_0)$ (all other $W$'s are empty)

[1] flag = false (flag is set to true each time a $W_{q, a}$ is changed)

[2] for all nonempty $W_{q, a}$ and
   for all $a \in \Sigma$ such that $M$-transform$(q, a, a)$ is undefined do
     if $|\beta| \geq 2$
       then [2.1] (the stack size increases or is steady)
         if $G$-transform$(W_{q, a}, a)$ is undefined then noten1
         $T + G$-transform$(W_{q, a}, a)$
         $X + \phi$
       for all $(t, \gamma)$ in $T$
         if $|\gamma| \leq |\text{rem}(t, \beta)|$ then noten1
         $k + |\gamma| - |\text{rem}(t, \beta)|$
         if $\text{rem}(k, \gamma) \neq \text{rem}(t, \beta)$ then noten2
         $X + X \cup \{(t, \text{first}(k, \gamma))\}$
       if $X$ is not a subset of $W_p, \text{first}(t, \beta)$, add $X$ to this set; flag = true
     else [2.2] (the stack size decreases)
       $k + \ell - |\beta|$
       for all $u \in \Phi_k(q, a)$ do
         [2.2.1] $V + \phi$
           for all $(t, \gamma) \in W_{q, a}$ do $V + V \cup \{(t, \gamma u)\}$
           if $G$-transform$(V, a)$ is undefined then noten1
           $T + G$-transform$(V, a)$
           if $T$ is not a subset of $W_p, \beta u$ then add $T$
           to this set; flag = true
     [3] if flag = true then go to [1]

[4] stop
A.3 Sketch of a proof and comments

For the time being, let us assume first, that, the algorithm stops, second, that the recognizer is $E_n(l)$, i.e. that the primitives noten are never executed (this will be explained later). We claim, then, that when the algorithm terminates $W_{q,a} = U_{q,a}$ for all reachable $(q,a)$.

The organization of the proof is the same as for the algorithm which constructs uncertainty zones for regular languages (IV.2). To prove $W_{q,a} \geq U_{q,a}$ is easy. To prove $W_{q,a} \leq U_{q,a}$ is done by induction. We just give an outline of how this proof can be conducted.

Given some $W_{q,a}$ (assumed to be a subset of $U_{q,a}$) and some symbol $a$ such that $a : (q,a) \vdash (p,b)$, the core of the algorithm computes a new set $W_{q,b'}$.

Now, $a : (q,a) \vdash (p,b) \Rightarrow \forall \xi$, $a : (q,a\xi) \vdash (p,b\xi)$.

One of two cases holds:

\[ [2.1] \ |B| \geq \ell \]

$W_{p,\text{first}(l,b)}$ is computed in the following way:

$W_{q,a} \leq U_{q,a} \Rightarrow V\xi$, $W_{q,a\xi} \leq U_{a,a\xi}$

$T$ is the image of $W_{q,a}$ by $G$ over $a$.

Clearly, $(t,\gamma) \in T \Rightarrow V\xi$, $(t,\gamma\xi) \in U_{p,b\xi}$

Let $b' = \text{first}(l,b)$, $b'' = \text{rem}(l,b)$ and $k' = |b''|$

The last $k'$ elements of $\gamma$ are identical to $b''$ (since it is assumed that noten is not executed). Therefore,
\[(t, \text{first}(k, \gamma)) \in U_{p, \beta}^r. \text{ That is } T \subseteq U_{q, \beta}^r \text{ and } W_{q, \beta} \leq U_{q, \beta}^r.\]

\[\text{[2.2]} \quad |\beta| < \ell\]

Here, instead of truncating configurations, we pad them with adequate symbols. A proof would be similar to the one for \([2.1]\).

We assumed that the algorithm stopped and that no primitive noten1 is executed. Let us examine these assumptions.

First, noten1 is a primitive which indicates that the recognizer is not \(E_n(x)\) for the given \(n\) and \(x\).

notations1 when executed means:

\[\exists(q, \alpha), \exists(t, \gamma) \in U_{q, \alpha} \text{ such that } |\gamma| < |q| - \ell.\]

The computation cannot proceed.

notations2 when executed means:

\[\exists(q, \alpha), \exists(t, \gamma) \in U_{q, \alpha} \text{ such that } \gamma \text{ differs from } \alpha \text{ by more than } \ell \text{ top-most elements of } \alpha. \text{ The computation can, however, proceed yielding sets } W's \text{ which are not uncertainty zones, but approximations to these zones. We discuss this idea at the end of the section.}\]

Now, does the process always terminate? In fact, as the algorithm stands now, the answer is "no." If the size of some \(U_{q, \alpha}\) is infinite the process does not stop. However, if one can prove that any \((t, \gamma)\) in any \(U_{q, \alpha}\) is such that
\[ |\gamma| \leq |\alpha| + \ell' \] for some constant \( \ell' \) then the process must terminate because it computes sets \( W's \) of increasing size and the \( W's \) are subset of

\[ Q' = \bigcup_{i=0}^{2+\ell'} \Gamma \]

which is a finite set.

We conjecture that, in the general case, it is not decidable whether such a bound \( \ell' \) exists. However, we shall see that for most programming languages the sizes of the uncertainty zones are finite. If one is unable (or unwilling) to prove in a particular case that a bound exists, one may guess a bound \( \ell' \) and check its validity by running the previous algorithm with the following block [3]:

\[ [3] \text{ if for any } (t, \gamma) \text{ in any } W_{q, \alpha}, |\beta| \geq \ell + \ell' \]

then \( \ell' \) not a bound

else if flag = true then go to [1].

This modified algorithm always stops.

Remarks:

1) This algorithm has been written with clarity as the main purpose (we do not claim that this purpose has been fully realized). It could be made more efficient. In particular, the elements of each set could be tagged by "old," "being treated," or "new." The function G-transform would ignore "old" elements to avoid duplication of work. Every element
added to a set would be tagged by "new." At the end of each pass through block [2], "being treated" elements would be reclassified "old" and the "new" ones would be changed to "being treated."

2) It may happen that for some \( q \) and \( \alpha' \), \(|\alpha'|<\xi\), there exists a set \( U_{q,\alpha} \) such that:

\[
\forall \xi, \ |\alpha'|+|\xi|\geq \xi, \ U_{q,\alpha',\xi} = \{(t,\gamma\xi) \mid (t,\gamma) \in U_{q,\alpha'}\}.
\]

Thus, instead of storing \( U_{q,\alpha',\xi} \) for all \( \xi \), it is only necessary to store \( U_{q,\alpha'} \) once. It is even possible to compute \( U_{q,\alpha'} \) directly but the algorithm is fairly complex.

A strategy for non \( E_n(\xi) \) parsers

In V.3.3, we suggested possible strategies for non \( E_n(\xi) \) parsers. In particular, we suggested using truncated configurations, even though they do not contain all the information needed to compute the uncertainty zones.

As noted earlier, the algorithm can be used to compute sets \( W \) if \texttt{notenfl} is never executed. \( W_{q,\alpha} \) can be used as an approximation of the uncertainty zone associated with the truncated configuration \((q,\alpha)\). We have shown (V.4.2) that for practical programming languages

\[
(p,\beta) \in U_{q,\alpha} \Rightarrow |a| - k \cdot n \leq |\beta| \leq |a| + k \cdot n
\]

for some constant \( k \) dependent on the recognizer. This
means that, if $L$ is chosen large enough ($L > k.n$), notenda
is never executed. The above approximated uncertainty zones
can, therefore, be computed.

To use these latter zones is approximatively equivalent
to correcting only the parts of the program which correspond
to the inner levels of nesting with respect to the level at
which the error is detected. This is quite a natural approx-
imation.
REFERENCES


<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Year</th>
<th>Title</th>
<th>Source</th>
</tr>
</thead>
</table>