A NATURAL LANGUAGE PARSER WITH INTERLEAVED SPELLING CORRECTION SUPPORTING LEXICAL FUNCTIONAL GRAMMAR AND ILL-FORMED INPUT

BY

MOHAMMAD ALI ELMI

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate School of the
Illinois Institute of Technology

Approved ______________________
Advisor

Chicago, Illinois
December 1994

ACKNOWLEDGMENT

I would like to thank my research advisor, Professor Martha Evens for her constant guidance, endless patience, and constructive criticism throughout this work. Her vast knowledge is superseded only by her limitless enthusiasm. It has been a pleasure working with her.

I am grateful to the members of my thesis committee: Prof. Ilene Burnstein, Prof. Jim Kenevan, and Prof. John Snapper for taking time from their busy schedules to support this effort.

My colleague Michael Glass deserves special thanks. I had the pleasure of working with him on the issues related to the grammar and the lexicon.

I am indebted to my wife, Maria, for being so understanding of my long working hours. She took good care of our two daughters with less and less help from me. My daughters Sheida and Roxana had to give up a lot for this work. Sheida, for staying away from the PC and putting up with the short bedtime stories that got shorter and shorter as this work was coming to an end. Roxana, had to give up a lot of play time with me, because I was not available.

M.E.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

List of Tables

Table Page

Figure Page

LIST OF Figures

Figure Page

ABSTRACT

This thesis describes the design and development of a parsing algorithm for context-free grammars. The parser is used in an intelligent tutoring system for cardiovascular physiology, called Circsim-Tutor, that assists first year medical students to learn the causal relationships between the parameters of the circulatory system and to solve problems resulting from perturbations to the system.

The parsing algorithm is a mixture of top down parsing techniques and bottom up parsing methods combined with chart parser techniques. The result is a fast parser that handles ambiguous grammars. The parser avoids the limitations of top down parsers, i.e. it can handle left recursive nonterminals. It can handle grammars with thousands of rules and nonterminals.

Parsing natural language requires the parser to be able to deal with syntactic ambiguity as well as lexical ambiguity. Integrating the parser with the Circsim-Tutor system requires the parser to have a rapid response, to be able to work with a large lexicon, and, more important, to interpret ill-formed student inputs such as spelling errors, fragments and abbreviations. There is absolutely no limit on the size of the grammar, nor on the size of the lexicon.

An interactive natural language environment requires that the errors be detected and corrected without user intervention. By interleaving the parser with the process of spelling error correction, and extending the three way match algorithm to four way match, the thesis introduces an efficient, intelligent, and reliable spelling corrector. This method, which is based on the goal of using the context for spelling correction, outperforms our previous system.

The lexical analyzer and the parser work hand in hand: the lexical analyzer gets the next input word and passes it to the parser. As the parser is processing the next word of the input, all applicable semantic checks are performed on the input word.

Enforcement of semantic rules is based on Lexical Functional Grammar theory. The grammar and the lexicon are augmented with LFG annotation rules. After successfully parsing the user input, the system builds the LFG f-structure, which is turned into a logical form to be used by the input understander of the tutoring system.

CHAPTER I

INTRODUCTION

1.1 Introduction

The main objective of a natural language processing system is to communicate with people in a natural and comfortable way. As a result, the natural language understanding system treats the user input as a meaningful piece of language and not as a meaningless string of words/characters. Compared to the conventional method of using a menu based dialog system, natural language interfaces have overcome the communication barrier. For one thing, there is no need to read pages and pages of user manual to use a system that has a natural language interface.

Intelligent tutoring systems have a high volume of information exchange between the system and the user. As intelligent tutoring systems grow and become more robust, they need natural language dialogues with users [Fox, 1991].

We are developing an intelligent tutoring system (ITS) that will assist first year medical students to learn the causal relationships between the parameters of the circulatory system and to solve problems resulting from perturbations to the system. One component of the system is the input understander, which is based on natural language. Its main job is to parse the student input, and put it into a logical form. Parsing efficiency is extremely important to the system.

The main distinction between a natural language such as English, and a formal language such as "C", is that in a formal language the grammatical rules are well defined and strictly enforced, whereas, in a natural language, the rules are not always well defined and the enforcement is relaxed most of the time. For example, many incomplete sentences are communicated amongst people that are perfectly understood, while in a formal language, sentences must be complete, and syntactic rules must be obeyed. Natural languages also grow and change over time.

The inherently ambiguous nature of the English language dictates that the parser must be able to reuse the segments that are already parsed. Also, in an interactive environment, spelling errors occur very frequently. The system must be able to automatically find a replacement candidate for the misspelled word, in contrast to word processing systems that offer a list of replacement words and ask the user to select one of them.

This thesis introduces an efficient parsing algorithm for natural language processing. I believe a natural language parser in an integrated environment should be able to:

- Deal with all sorts of ambiguous sentences, as English grammar is full of ambiguity.
- Work with input with multiple parts of speech, e.g., 'that' is a determiner, a pronoun, ...
- Enforce the semantic rules. Specifically, as soon as any semantic rule is violated in a branch, the parsing of the branch should be terminated.
- Handle all sorts of ill-formed input, fragments (ellipses), spelling errors.

1.2 Outline of This Thesis

This thesis introduces an efficient method for lexical, syntactic, and semantic analysis of the user input. Interleaving the lexical, syntactic and semantic analysis of the input, gives each module access to the context available in other modules.

The lexical analyzer and the parser work hand in hand: the lexical analyzer gets the next input word and passes it to the parser. As the parser is processing the next word of the input, all applicable semantic checks are performed on the input word. The advantage of interleaving the three modules (lexicon, syntax and semantics) is very early detection of errors, be it spelling errors, syntax errors, semantic errors, or a simple sentence fragment.

Another advantage of this method is that it allows the modules to use each others' context for error recovery. For instance, the spelling correction is not done in isolation. After the initial search for the possible replacement candidates for the misspelled word, the parser can aid in the elimination of irrelevant words. If there still remains more than one candidate word, the semantic rules can help eliminate irrelevant words.

The lexicon organization was set up to facilitate spelling correction, phrase recognition and abbreviation entry. A hash table guarantees fast access to the entries. Because of the rich set of lexical entries and the lexical ambiguity in a natural language environment, the lexicon is organized in such a way that there is no limit on the number of lexical entries. Each lexical entry consists of the actual word string, its part of speech, and a set of structures including Lexical Functional Grammar (LFG) functional schemata and case frame information. Aside from normal entries, a lexical entry can consist of several words (a phrase), or it can simply be an abbreviation of another entry. A single word can possess multiple parts of speech and even multiple semantic structures.

The parsing algorithm is a mixture of top down parsing techniques and bottom up parsing methods combined with chart parser techniques. The result is a fast parser that handles ambiguous grammars. The parser does not have the limitations of the original top down parsers [Backhouse, 1979]. For example, it can handle left recursive nonterminals. The parser meets the challenges facing a natural language parser in an integrated environment. Namely, it:

- demonstrates speed and space efficiency.
- allows augmentation of semantic rules in the grammar. The parser enforces the semantic rules. Also, as soon as any semantic rule is violated in a branch, the parsing of the branch can be terminated.
- handles all sorts of ill-formed input such as fragments and ellipses.
- recognizes the phrases within a sentence.
- accepts input with multiple parts of speech.
- uses syntactic/semantic information to help correct the spelling errors.
- deals with all sorts of ambiguous sentences, as natural language is inherently ambiguous.

An interactive natural language environment requires that the errors be both detected and corrected without user intervention. By interleaving the parser with the process of spelling error correction, and extending a well known spelling correction algorithm, the thesis introduces an efficient, intelligent, and reliable spelling corrector. This method, which is based on the goal of using the context for spelling correction, is very powerful. After all, the best spelling correctors, human beings, use the context for more complex spelling errors.

Enforcement of semantic rules is based on the concepts of Lexical Functional Grammar [Bresnan et al., 1982]. The grammar and the lexicon are augmented with LFG annotation rules. After successfully parsing the user input, the system builds the LFG f-structure, which is turned into a logical form to be used by the input understander of the tutoring system.

Another goal of this research is to show that it is possible to do a good job with a top down approach to parsing natural language. Of course, the top down algorithm must be extended. I will show that the extended parser, if not better and faster, is as good as the bottom up technique.

Handling of fragments and ellipses requires no special work for the system. It does it naturally. No added productions or new nonterminals are needed. Once a sentence cannot be fully recognized the parser tries to match the input with all nonterminals of the grammar.

The two basic building blocks of my system, the grammar and the lexicon are constantly changing. Since we are experimenting with various grammar productions, the fact that the parser is table driven and operates independent of a particular grammar has proven to be a very nice feature of the system. The lexicon itself is also constantly being modified to cover all aspects of our system, not only for the purposes of the input understander, but also, for the text generation module.

To cope with the frequent changes in the grammar and the lexicon, a parser generator automatically generates parse tables. I have written a lex-builder program that reads a file of raw lexical entries and generates the final lexical organization. These two programs, in combination with the fact that the parser is table driven, have enabled us to concentrate on the essence of the incremental task of lexicon building and adapting the grammar to cover our sublanguage domain.

1.3 Summary of the Research

The main objective of this research was to come up with a reliable and fast parser to parse natural language with no restrictions on grammar ambiguity. Furthermore, there should not be any limitation such as the size of the grammar or the size of the lexicon. The second objective was to have a reliable and automatic spelling corrector.

1.4 Organization of the Thesis

Chapter II contains a review of the literature. An overview of parsing methods follows with a description of the current spelling correction algorithms. A brief overview of the lexical analysis methods is also presented.

Chapter III describes the syntax analysis process. A detailed description of my parsing method is presented. The method of creating the logical form from the syntax tree is based on concepts of the theory of Lexical Functional Grammar. The parsing algorithm is presented together with a detailed example.

Chapter IV covers the parser generator and the structure of the parsing tables.

Chapter V covers the spelling correction algorithm and the interaction amongst the three modules: the lexical analyzer, the parser and the semantic analyzer.

Chapter VI covers the lexical structure and the building process of the lexicon.

Chapter VII covers the lexical analysis of the system. An algorithm for recognizing word phrases is illustrated together with a detailed example.

Chapter VIII examines the interaction amongst the system components.

Chapter IX summarizes the research and gives directions for future study.

CHAPTER II

REVIEW OF RELEVANT RESEARCH

This research involves three areas: 1) a parser for natural language, 2) spelling corrections, and 3) lexical analysis of the input words.

In the introductory chapter, we pointed out that natural language is ambiguous. The first section of this chapter briefly examines syntactic ambiguity, and the following three sections review the literature in the areas of parsing, spelling correction and lexical analysis.

2.1 Syntactic Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ambiguous [Aho et al., 1986, p. 171]. Earley [1970] defines ambiguity as follows:

- The degree of ambiguity of a sentence is the number of its distinct derivation trees.
- A sentence is unambiguous if it has degree 1 of ambiguity.
- A grammar is unambiguous if each of its sentences is unambiguous.

Backhouse [1979, p. 25] states that the property of ambiguity is (usually) a property of the grammar rather than the language. Aho and Ullman [1972, p. 205] state that a context free language is called inherently ambiguous if it has no unambiguous context free grammar. Problems of inherent ambiguity do not rise in programming languages.

Syntactic ambiguity is when a given sentence can be derived in more than one way by the rules of the grammar. In other words, a given sentence has more than one analysis. Sager [1981, p. 5] gives an interesting example of syntactic ambiguity:

They are visiting relatives.

One reading here has the relatives doing the visiting, and in another those who are doing the visiting are different from the relatives. In the second interpretation, 'are visiting' is a verb, whereas in the first interpretation, 'visiting' is a modifier of the 'relatives'.

A simpler example of ambiguity is in: " *John saw the car walking down the street* . Here, 'walking down the street' could modify the car, rather than John, aside from the intrusion of our world knowledge", [Berwick, 1987, p. 164].

Tomita [1986, p. 5] mentions that although it is ideal that the parser produce only one parse tree for the given ambiguous sentence, in some situations this is not possible, as some sentences are absolutely ambiguous. That is, even humans cannot disambiguate them. The parser must produce all parse trees of the sentence and leave it to the rest of the system to disambiguate the given parses.

In an integrated parsing environment, such as our Circsim-Tutor System [Kim et al., 1989], ideally, the parser should choose one syntactic interpretation so that the rest of the system can respond to input from the user. One way to do this is to follow the type of parsing process that people use. The fact that people don't usually recognize an ambiguity but pick a single interpretation, suggests that some analyses are favored over others [Allen, 1987, p. 161]. In our system, at this point, we take the first branch that consumes the maximum number of inputs, and preferably the whole sentence. Of course, the chosen branch must satisfy all semantic rules, including LFG f-structure constraining rules.

2.2 The Parser

A lot of work has been done in the area of compiler parsing techniques. Compilers have been parsing computer programs for decades. What important problems are left in parsing natural language? A simple answer is handling its ambiguity in an intelligent (and efficient) way.

A natural language parser differs from a programming language parser in the following broad aspects:

1. Ambiguous grammar: Natural languages are inherently ambiguous. Most programming languages have one derivation for a given statement, whereas many natural language sentences have many valid derivations.
2. Multiple parts of speech: The input words may have more than one part of speech, e.g., 'that' is a determiner, a pronoun, etc. Most programming languages only allow one lexical category per symbol. To make the matter worse, a word can have several *senses* for a single part of speech.
3. Ill formed input: while computer programs require the programmer to follow stringent syntactic rules, fragments and ellipses are nothing new to a natural language.

Ralston and Lee [1983, p. 1105] define parsing as "the process by which the phrases in a string of characters in a language are associated with the component names of the grammar that generated the string. A parse is often shown as a tree." In natural languages, the syntax of a language, its grammar, defines the valid relationship between the elements of the language. Semantics is the definition of meaning that is prescribed by the speaker. Allen [1987, p. 6] states that pragmatic knowledge concerns how sentences are used in different contexts and how context affects the interpretation of the sentence. Semiotics, the theory of symbols, is concerned with these three aspects of symbols [Lee, 1983a, p. 1233]. Webster's dictionary [1988] defines semiotics as "A general philosophical theory of signs and symbols that deals especially with their function in both artificially constructed and natural languages and comprises the three branches of syntactics, semantics, and pragmatics."

A programming language must be unambiguous, or, at least, any ambiguities in the language must be immediately evident and easily avoided [Backhouse, 1979, p. 14]. The set of symbols available for sentence composition, in computer language, is highly restricted. Problems of inherent ambiguity do not rise in programming languages [Backhouse, 1979, p. 25]. Natural language, on the other hand, is inherently ambiguous, and the set of symbols is very large. A subtle distinction between programming languages and natural languages is that in parsing a natural language, the input length is typically small (10-12 words) and the grammar size very large (hundreds or thousands of rules and symbols) [Carol, 1994, p.292].

Whereas, in parsing a programming language, the input length is usually large (thousands or tens of thousands of words), the grammar size is usually on the order of hundreds of rules.

This may bring the application of compiler parsing techniques to parse natural language into question. But, we will see that this is not a big problem. With existing systems such as Tomita's parser [Tomita, 1987], the question is not whether these methods are applicable or not. The question is how to efficiently utilize these techniques for natural language parsing.

Although there are a few methods of parsing natural language, the parsers developed for programming languages show more promise. Computer scientists have spent many years coming up with efficient parsers for programming languages. It makes sense to examine their application to parsing natural language.

Aho et al. [1986, p. 160] mentions that there are three general types of parsers for context free grammars: universal parsers based on dynamic programing such as the Cocke-Younger-Kasami algorithm [Aho and Ullman, 1972], can parse any context free language. These methods, although powerful, are too inefficient. The two methods bottom up, and top down, are the best known parsing techniques in the area of compiler parsing. Section 2.2.1 describes bottom up and top down parsers in detail.

A very famous universal context-free parser is *Earley's algorithm*, developed by Jay Earley [1970]. Earley's algorithm is the method of choice for universal parsers. Aho and Ullman [1972, p. 314] state that Earley's algorithm is superior to the Cocke-Younger-Kasami algorithm.

Earley's algorithm can be viewed as an interpretive version of LR parsing. Although it is more general than the LR-style technique, it is slower. LL and LR parsers work in linear time and space. Earley's algorithm sometimes requires cubic time, and quadratic space, though this nonlinear performance usually occurs while parsing using grammars beyond the capability of ordinary parsers [Fischer and LeBlanc, p. 205]. Tomita [1986, p. 2], refers to universal context-free parsers as general parsers. The fact that these algorithms are too inefficient is not surprising, because they are designed to handle difficult phenomena in context free languages, such as ambiguity and cycle. In other words, these algorithms are efficient for hard grammars by sacrificing efficiency in easy grammars.

Tomita [1986, p. 6], defines the two terms ambiguity and cycle as: a grammar is infinitely ambiguous if it produces an infinite number of parses with some input sentences. A grammar is cyclic, if there exists a nonterminal that can be reduced to itself. The two phenomena are equivalent: a grammar is infinitely ambiguous if and only if it is cyclic.

2.2.1 Bottom Up and Top Down Parsers. Top down parsing methods attempt to construct a parse tree for an input string beginning at the top (root) and working to the bottom (leaves). Bottom up parsers try to construct a parse tree for an input sentence beginning at the bottom and going to the top.

To be brief, we could say that the fundamental concern of a top-down parser is deciding which production of a particular nonterminal should be expanded, whereas, the fundamental concern of a bottom-up parser is deciding when what looks like the right-hand side of a production can be reduced to its left-hand side. Considering the fact that more than one production may have the same right hand side, this is not a trivial task {Fischer and LeBlanc , 1988].

Top down parsers can be constructed automatically using the class of LL(1) grammars. Aho et al. [1986, p. 191] define LL(1) as: A grammar whose parsing table has no multiply-defined entries is said to be LL(1). The first 'L' in LL(1) stands for scanning the input from left to right, the second 'L' for producing a *leftmost* derivation, and the '1' for using one input symbol of look-ahead at each step to make parsing action decisions.

Similarly, the term LR, in bottom up parsers, is defined as follows. The first 'L' in LR stands for scanning the input from left to right, and the 'R' for producing a *rightmost* derivation. In a rightmost derivation, the rightmost nonterminal is replaced at each step.

The concepts underlying LL parsers are very easy to visualize and understand. The concepts underlying LR parsers are rather more complicated. It is very easy to hand write a top down parser in the form of recursive descent an LL(1) method. An LR parser, on the other hand, requires a tool to generate the parse tables for any relatively large grammar. It must be mentioned that in an environment where the grammar changes a lot, it is wise to have a tool to generate the parser, regardless of the chosen parser technique. Aho and Ullman [1972, p.449] state that LL(1) grammars are particularly attractive for practical use. For each LL(1) grammar we can find a left parse, which is advantageous for translation purposes.

Bottom up parsing can handle a larger class of grammars with fewer restrictions placed on them. However, LR parse tables are larger than LL parse tables and LR parsers tend to be slower than LL parsers [Holub 1990, p. 337]. LL parsing is particularly attractive on account of its naturalness [Backhouse 1979, p. 92]. LR does not offer the rich structure that enables an LL parser to incorporate error recovery. Left recursion must be eliminated in the grammar, otherwise, the grammar is not an LL grammar. Left recursion presents no particular problems to LR parsers.

Although LL has a deciding edge on space efficiency, both methods are time efficient. Unfortunately, there is little published research that compares the performance of the two methods for a given grammar. There is no doubt that both are efficient and for all indications, there is not a big difference in time efficiency between the two. Holub [1990, p. 337] states that LR parsers tend to be slower than LL parsers while Bagley [1992, p. 6] claims that LL and LR have essentially the same time efficiency.

The LALR (look-ahead-LR) technique is often used in practice because its table is considerably smaller than the canonical LR tables, yet, it is powerful enough to cover most common syntactic constructs of programming languages. For a comparison of parser size, the LALR table typically consists of several hundred states for a language like Pascal, while the canonical LR table typically has several thousand states for the same size language [Aho et al., 1986, p. 236].

Backhouse [1979, p. 187], mentions that Heilbrunner [1] presents an LL(1) grammar, that is not LALR(1), contradicting a claim made by Aho and Johnson [1974].

We started the chapter by mentioning the differences between the two environments (programming language, natural language). It is clear that the existing methods need to be extended to parse natural language. One of the best natural language parsers with a wide acceptance is Tomita's parser [1987]. Tomita's parser is an extended LR parser. It is efficient and can handle all natural language parsing requirements. Because of its importance, a section is dedicated to this parser.

A detailed discussion of universal parsers and other parsing methods can be found in [Aho and Ullman, 1973]. A history of natural language processing can be found in [Shapiro, 1983]. A review of programing language parsers, as well as natural language parsers, can be found in [Bagley, 1992].

2.2.2 Ellipsis. When elements of a sentence that are predictable from context are omitted, we have an elliptical sentence [Quirk et al., 1985, p 82]. Read et al. [1988, p. 427] defines ellipsis as: "an incomplete sentence that gets its meaning only in context." For example, the answer to the question: "Are they paying you for the work?" could be: "Yes, they are." which is actually: "Yes, they are 'paying me for the work'" which is an unreduced answer.

The sentence is the highest ranking unit of the grammar [Quirk et al., 1985, p 47]. The parser's aim is to recognize the full sentence by trying to parse the input into the start symbol of the grammar. When a full sentence cannot be recognized, the parser must assume that it is encountering an elliptical sentence and should try to find all nonterminal symbols of the grammar that can parse the input.

A good description of elliptical sentences in human computer dialogues is found in [Frederking, 1988] and [Lee, 1990]. A comprehensive study of ellipsis can be found in [Quirk et al., 1985].

2.2.3 Syntactic Error Recovery. Error recovery is an important part of syntax analysis and it makes a natural language parser far more valuable. The main reason for the lack of this functionality in most natural language parsers is that it is too hard to come up with error recovery since the language is inherently ambiguous.

In an unambiguous environment, the parser can use the syntax information to resume parsing after considering insertion of some token or skipping the text. This action could possibly add a few branches to the parse tree at the point of error, since, the parser has to examine all options: skipping some tokens or adding some tokens.

Because of the ambiguity of English grammar, there is normally more than one syntax derivation for a sentence. Addition of error repair further increases the number of derivations. In order to have successful error recovery, the parser has to be able to efficiently manage all branches, by using semantic rules to reduce the number of possible parse trees. The underlying parser algorithm should be flexible

[1] Heilbrunner, S., (1977) Using Item Grammars to Prove LR(K) Theorems, Bericht Nr. 7701 Holschule Bundeswehr Munchen, Fachbereich Informatik.

and must be equipped for error recovery. Special care must be taken when a section of input with a syntax error must be reparsed.

Of the two techniques, bottom-up and top down parsing, top down can do a better job of error recovery. Backhouse [1979, p. 157], mentions that as a practical alternative to the LL technique, LR parsing suffers because it is unnatural and does not offer the rich structure that facilitates the incorporation of error recovery and code generation in an LL parser. Error recovery in a bottom up parser is notoriously difficult, according to Holub [1990, p. 348]. Having an LL parse stack provides better error recovery possibilities because we know where we are and where we are going. Whereas in LR parsers, we don't always know for sure where we are going until we get there. In other words, bottom up parse stacks contain information on what has already been seen. The power of the LR parsing method comes precisely from the ability to delay this decision.

It is very common for the users of a natural language system to make spelling errors. Most researchers have concentrated on spelling error correction and omitted syntactic error recovery all together. Although I agree that syntactic error recovery is not as important as spelling error correction, as syntactic errors do not happen as often as spelling errors, I believe it is a useful addition to the system. Section 2.3 provides a detailed analysis of spelling correction methods.

2.2.4 Chart Parser. As mentioned, the parser must follow all branches of the syntax tree when it encounters ambiguity. The backtracking parser does this with a brute force approach; it takes every possible branch and parses it until it reaches failure or complete success.

The chart parser [Allen, 1987, p. 102] allows one to save the result of parsed segments in a chart. As new branches are ready to be parsed, segments that have already been saved in the chart are reused instead of reparsed.

A top down parser does not put together combinations that would not fit into the overall structure of a sentence. A bottom up parser does not build structures that will fail to complete their later parts. Chart parsing technology is intended to be a more general approach that avoids the commitment of strict bottom up and top-down methods [Lee, 1990].

2.2.5 Tomita's Parser. Because of the ambiguity in any natural language, in many situations there is more than one derivation tree for a given sentence. A parser, one way or another, has to take all appropriate branches to generate all parse trees for a given sentence, nondeterministically or at best nearly deterministically. Once a parse table has multiple entries, deterministic parsing is no longer possible and some kind of nondeterminism is necessary [Tomita 1986 p. 13]. One way to do this, at any point when there are multiple choices, is to allow the parser to take all choices and continue. The efficiency of a parser is mostly determined by how it handles multiple branches of the parsed tree and avoids reparsing.

Tomita's parser uses a generalized LR algorithm that allows shift/reduce conflicts. An LR parser cannot handle this type of conflict. Tomita relaxes this restriction, and, by transforming the parse stack into a graph-structured stack, his parser can take more than one action. The graph-structured stack allows the parser to maintain multiple parses without parsing any part of the input twice. The graph-structured stack looks similar to a chart. Unlike the standard LR parsing algorithm, it can handle arbitrary context-free grammars, including ambiguous grammars.

The *graph-structured* stack was created to cover a wide range of parsing problems that is required to parse an ambiguous grammar. The first step for a parser to handle shift/reduce conflict problems is to have a multiple number of stacks. Each stack takes one possibility, which is referred to as a *Stack List*. Having multiple stacks causes a lot of reparsing of the same segment of the input. He transforms the stack list into a *tree-structured* stack by the following steps:

- Combine the stacks that have common rightmost symbols.
- Split the stack when popping (reduce action).

There still remains the problem of copying. That is, any time the stack is split, the whole stack has to be copied. This problem is avoided by using a *graph structure* in which common segments are shared. When two or more trees have common subtrees, the subtrees are represented once.

There is basically no restriction on this parser as far as ambiguity of the grammar. It produces all parse trees for ambiguous sentences.

2.2.6 Lexical Functional Grammar. Bresnan and Kaplan [1982] introduced Lexical Functional Grammar (LFG) to model syntactic processes by means of psychological studies of competence-based models of language acquisition, comprehension, and production. Lexical Functional Grammar uses two levels of syntactic representation for a given input. The first form is Constituent Structure or c-structure and the second form is Functional Structure or simply f-structure. Kaplan [1989, p. 1] defines the two structures as follows:

> The c-structure is a phrase-structure tree that serves as the basis for phonological interpretation while the f-structure is a hierarchical attribute-value matrix that represents underlying grammatical relations. The c-structure is assigned by the rules of a context-free phrase structure grammar. Functional annotations on those rules are instantiated to provide a formal description of the f-structure, and the smallest structure satisfying those constraints is the grammatically appropriate f-structure.

The c-structure represents the order arrangement of words and phrases in the sentence while the f-structure explicitly marks its grammatical functions (subject, object, etc.) [Kaplan et al. 1989 p. 273]. The ability to use two structural representations, c-structure and f-structure, allows one to deal with the fact that there are different kinds of informational dependencies among the parts of a sentence. The first level of syntactic representation, c-structure, is very similar to the regular syntax tree and is referred to as a *phrase structure*. The second level of syntactic representation, f-structure, is made of universal formats to represent grammatical relations. Once the c-structure is built, construction of the f-structure begins.

The correspondence between c-structure and f-structure was first sketched by Bresnan and Kaplan [1982]. There is a piecewise function that maps from the nodes in the c-structure to the units of the f-structure. The piecewise function, Φ, corresponds to a map between c-structure and f-structure. The *c-structure-to-f-structur*e correspondence, Φ, is a many-to-one function [Chang, 1992, p. 14].

Figure 2.1, borrowed from Kaplan et al. [1989], shows the concept of a structural correspondence in which structures of different types can be set into correspondence by a piece-wise function for the sentence:

The baby fell.

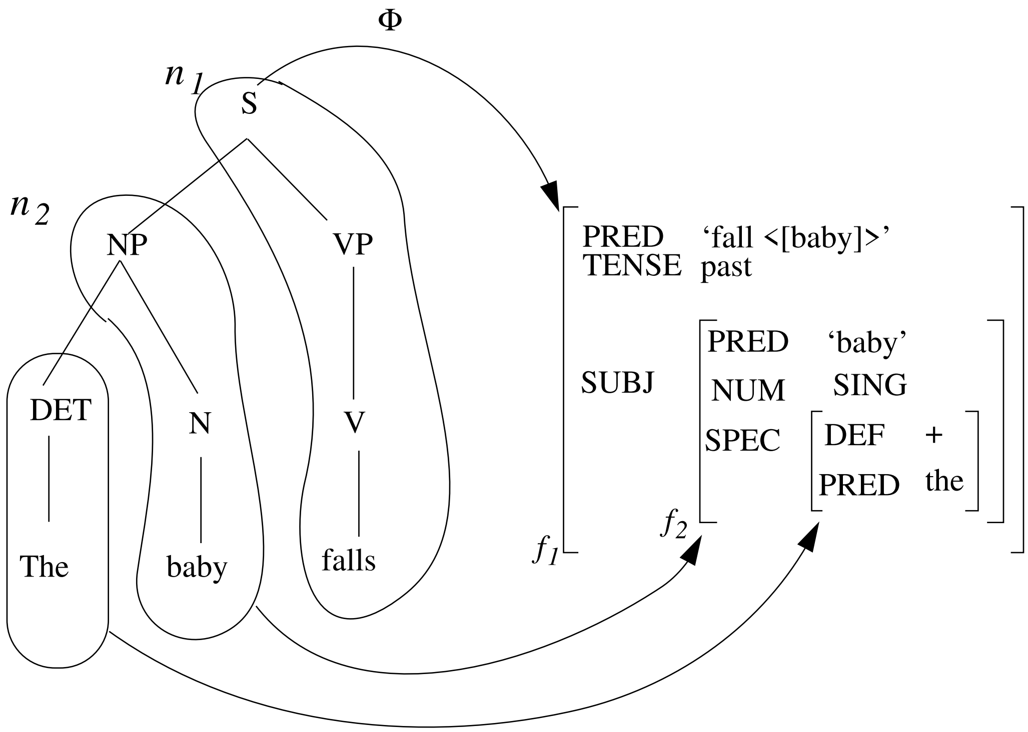


Figure 2.1 A Structural Correspondence.
Adapted from Kaplan et al. [1989, p. 273]

In Figure 2.1, the c-structure appears on the left, the f-structure on the right. As stated in Kaplan et al. [1989] the c-structure-to-f-structure correspondence, Φ, is shown by the linking lines. Note that the *c-structure-to-f-structure* correspondence function is a many-to-one function taking the S, VP, and V nodes all into the same outermost unit of the f-structure, *f1*.

Wescoat [1987, p. 27] states that in order for a sentence to be predicated grammatical by a Lexical Functional Grammar, it must satisfies the following two criteria:

1. The grammar must be able to assign it a constituent structure tree.
2. The grammar must assign it a well formed f-structure.

The definition and usage of three well-formedness conditions [Wescoat, 1987, p. 29] are looked at in detail in the following sections.

2.2.6.1 Uniqueness. The uniqueness condition guarantees that if there is a value 'x' associated with an attribute 'y' in the functional description 'fp', and the distinct value 'z' is also associated with attribute 'y', then the f-structure is ill-formed.

2.2.6.2 Completeness. Completeness states that if a grammatical function is mentioned in the predicate argument structure, then it must be represented in the f-structure. For instance, in the sentence: 'John kills' the lack of an object violates the completeness condition.

2.2.6.3 Coherence. Coherence is the inverse of completeness. It limits the occurrence of governable grammatical functions within the f-structure.

The f-structure builder is essential in checking the correctness of the sentence, beyond a simple syntax rule. For instance, in the sentence: "John falls Mary" the f-structure builder will generate the c-structure shown in Figure 2.2.

Construction of the f-structure reveals the problem that the verb 'fall' does not require an object, 'Mary'. Existence of the object 'Mary' violates the coherence requirements. Therefore, the sentence is considered ungrammatical.

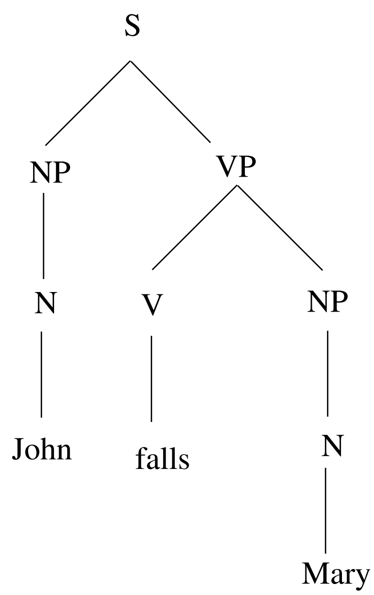


Figure 2.2 LFG C-structure for "John Falls Mary"

2.2.7 Comparison of LL and LR Parse Table Size. LR parse tables are generally larger than LL parse tables. In this section, we attempt to define the relationship between the grammar (its size, number of productions, number of terminal/ nonterminal symbols), and the size of the parse table. It is obvious that the bigger the grammar, the bigger the parse table, regardless of the parsing technique.

Holub [1990, p.368] gives the table sizes for an LALR(1) parser for the grammar of C: 182 productions, 50 terminals, and 75 nonterminal symbols. It generates 310 LALR(1) states. The parser table has 126 columns, terminals plus nonterminals plus the end of input marker, EOF, 50+75+1. Therefore, for 310 rows, we have a large number of states:

$$310 \times 126 = 39{,}060$$

Given the need for a two byte integer to hold state numbers, the size goes to 78,120 bytes. In comparison, the LL(1) parse table contains a select set of terminal symbols for each production. The LL(1) parse table entry needs a set containing valid terminals. Given 50 terminal symbols, the set must contain at least 50 bits, which is 7 bytes. Given 182 productions, the LL(1) parse table contains 182*7= 1274 bytes. Of course, removing left recursion and converting the grammar to be LL(1), increases the number of productions a little bit, but, the table sizes are not comparable. LL provides a substantial space saving.

YACC, Yet Another Compiler-Compiler [Johnson 1978], an LR parser generator, was developed for use in Unix systems and is widely used. Due to the size of LR tables, YACC, in the implementation on Sun Sparc stations, puts the following limits on the grammar size: 600 productions, with 200 terminals. Note that this restricts the usage of YACC, so that it cannot be used in generating a parser for a big language like VHDL [Baker, 1993], with 750 productions, 250 nonterminals, and 150 terminals.

There are ways to reduce the size of LR tables, as the tables are sparse, at the expense of CPU time. One method of compaction of LR parsing tables is to recognize that usually many rows of the action table are identical. There could be considerable saving of space, at little cost in time, to create a pointer for each state into a one-dimensional array. Then, the pointers for identical states point to the same location [Aho et al., 1986, p. 244].

2.2.7.1 LL (1) Parse Table. Associated with every production of the grammar, there is a set called the choice set (or select set). To start parsing, the start symbol of the grammar is pushed onto the stack. At every step, a nonterminal production is selected to be expanded when the next input symbol is in the select set of the production.

For choice set computation, we need to define FIRST and FOLLOW sets. The following definition is borrowed from [Aho et al., 1986, p. 188]. The ε denotes the null or empty symbol and $ is the end of file:

If α is any string of grammar symbols, let FIRST(α) be the set of terminals that begin the string derived from α. If $\alpha \stackrel{*}{\Rightarrow} \varepsilon$, then ε is also in FIRST(α).

Define FOLLOW(A), for nonterminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form. That is, the set of terminals a such that there exists a derivation of the form $S \stackrel{*}{\Rightarrow} \alpha A a \beta$ for some α and β. Note that there may, at some time during the derivation, have been symbols between A and a, but, if so, they derived ε and disappeared. If A can be the rightmost symbol in some sentential form, then $ is in FOLLOW(A).

In another manner of speaking, we might say that, given a context-free grammar with starting symbol <S> and grammatical symbol x, FOLLOW(x) is the set of input symbols that can immediately follow an x in an intermediate string derived from <S>. FIRST(x) is the set of those grammatical symbols that occur at the beginning of intermediate strings derived from x [Lewis et al., 1976, p. 437].

The choice set of each production $S \Rightarrow A_1 A_2 \ldots A_n$ is calculated as:

For i from 1 to n DO:

If A_i is terminal Then

add A_i to the choice set of the production and EXIT.
Else If A_i is a nonterminal Then
add FIRST(A_i) to the choice set of the production.
If A_i is not a null producer, (its first set does not contain ε) Then
EXIT.
End If
Else If the end of the production is reached Then
add the follow set of S to the choice set of the production.
End If
End For

The choice set needs to hold all terminal symbols of the grammar. If we use a bitmap representation, one bit is enough to represent a symbol (it is either in the choice set or not). Therefore, if p is the number of productions in a grammar and t is the number of terminal symbols in the grammar, and assuming there are 8 bits per byte, then, the table size will be:

$$\mathrm{LLsize} = p \times \frac{t}{8} \qquad (1)$$

This number is the accurate size of the parse table in an LL(1) environment. Note that $t/8$ is the upper bound value, i.e. 7/8=1.

2.2.7.2 LR (1) Parse Table. The LR(1) parse table is a two dimensional array in which there is a column for each terminal symbol and for each nonterminal symbol plus one reserved for the end of input. Under the terminal symbols, we put shift and reduce actions. Under the nonterminal symbols, we have *goto* actions. Usually, the state numbers are saved in the array. The number of rows is the number of states for the LR parser. Assuming that p is the number of productions in the grammar, t is the number of terminal symbols and n is the number of nonterminals in the grammar, each row of the parse table has length $(t + n + 1)$. The number of states is normally higher than 256. Therefore, we need at least two bytes per cell to save the state numbers. Thus, the number of bytes needed per state in an LR parse table is:

$$2 \times (t + n + 1) \qquad (2)$$

The next task is to calculate the number of states. Unfortunately, there are a lot of factors involved in the creation of states. Here, we do an analysis for the worst case. Let's see how many states are created for the following production:

$$N \rightarrow r_1\, r_2\, r_3\, r_4 \ldots r_n$$

In a simple case, one production produces n states where n is the number of right hand side symbols of the production. Note that we said "in a simple case." If a nonterminal has two or more productions that share some right hand side symbols, all of them generate only one state per shared symbol. Once the shared symbols are dealt with, then our formula is applicable. With this exception, we can come up with

Table 2.1 LR Table Generation

state	status	action
0	$N \rightarrow {}_\wedge r_1\, r_2\, r_3\, r_4 \ldots r_n$	with r_1 goto state 1
1	$N \rightarrow r_1\, {}_\wedge r_2\, r_3\, r_4 \ldots r_n$	with r_2 goto state 2
2	$N \rightarrow r_1\, r_2\, {}_\wedge r_3\, r_4 \ldots r_n$	with r_3 goto state 3
3	$N \rightarrow r_1\, r_2\, r_3\, {}_\wedge r_4 \ldots r_n$	with r_4 goto state 4
n-1	$N \rightarrow r_1\, r_2\, r_3\, r_4 \ldots {}_\wedge r_n$	with r_n goto state n
n	$N \rightarrow r_1\, r_2\, r_3\, r_4 \ldots r_n\, {}_\wedge$	N is recognized

the following formula for the number of states in an LR parse table:

$$\sum_{i=1}^{i=p} (\text{Number of right hand side symbols of production } i) \qquad (3)$$

where p is the number of productions in the grammar. We can see that the number of states grows rapidly when the number of productions grow. From (2), we have the number of bytes per state. Combining formulas (2) and (3) gives us the following formula for the number of bytes needed in an LR parse table:

$$LRsize = states \times 2 \times (t + n + 1) \qquad (4)$$

Comparing this with the LL(1) parse table size, defined in formula (1) on page 15, we get the following ratio:

$$\frac{states \times 2 \times (t + n + 1)}{\frac{p \times t}{8}}$$

after simplification:

$$states \times 16 \times \frac{(t + n + 1)}{(p \times t)}$$

Finally, the ratio of the table size for LR parse tables compared to LL parse tables is:

$$states \times 16 \times \left(\frac{1}{p} + \frac{(n + 1)}{(p \times t)}\right) \qquad (5)$$

where p is the number of productions in the grammar, t is the number of terminal symbols, and n is the number of nonterminals in the grammar.

We can simplify the formula for the parse table size of an LR parser, defined in (4), by using an average number of right hand side symbols. Let's assume that on the average, every production has r symbols in the right hand side. Therefore, the number of states is:

$$states = p \times r$$

Therefore, the simplified number of bytes needed in an LR parse table is:

$$LRsize = p \times r \times 2 \times (t + n + 1) \quad (6)$$

Comparing this to the LL(1) parse table size defined in (1)

$$\frac{LRsize}{LLsize} = \frac{p \times r \times 2 \times (t + n + 1)}{\frac{p \times t}{8}}$$

Simplifying the expression:

$$\frac{LRsize}{LLsize} = \frac{p \times r \times 16 \times (t + n + 1)}{p \times t}$$

Removing 'p':

$$\frac{LRsize}{LLsize} = \frac{r \times 16 \times (t + n + 1)}{t}$$

Finally, the simplified ratio of an LR parse table to LL parse table is:

$$\frac{LRsize}{LLsize} = 16 \times r + \frac{r \times 16 \times (n + 1)}{t} \quad (7)$$

It has to be emphasized, that, converting an LR grammar to LL grammar usually adds to the number of productions for left factoring symbols and the left recursion removal. Also, since the LR parse table is sparse, known techniques can be used to reduce the size. Of course, reduction of space comes with the expense in extra CPU time usage.

In the worst case, the number of states in an LR parser is **exponential** in the size of the grammar. Fischer and LeBlanc [1988, p. 213], give the following grammar that has $O(n^2)$ productions:

$$S \rightarrow X_i\, Z_i \qquad 1 \le i \le n$$

$$X_i \rightarrow Y_j\, X_i \mid Y_i \qquad 1 \le i, j \le n,\ i \ne j$$

and conclude that it has O(2^n) states. I wrote a program that generates the above grammar for a given value of n. Then, I ran the Unix YACC, a LALR(1) parser generator [Johnson, 1978], program on the generated grammar. The YACC program, on the Sun workstation, has the following limitation:

300 terminals, 300 nonterminals
600 grammar rules, 1000 states

For n=8, YACC gives up as the number of states grows beyond 1000. I took the YACC source code from the Sun machine, and modified it to raise the above limits. For n=10, it took over five minutes for the new program to generate 10,342 states. For n=11, it took over ten minutes for the new program to generate 22,651 states, before it crashed, while generating the action table (this test justifies the limits put on the original YACC program). Table 2.2 summarizes the result of the experiment, and shows the rapid growth of the number of states as *n* gets larger. The last two columns in the table give the size of LR using formula (4) on page 16 for the LR and the LL table using formula (1) on page 15. Note that this grammar is not LL(1) for n > 1, as 'y_i' needs to be factored out for S. However, my parser can be used to parse this grammar, with no problem.

Table 2.2 LR States for Fischer Grammar

n	No. of Terminals	No. of Nonterms	No. of Prod	No. of States	LR Size	LL Size
1	4	2	3	5	35	3
2	6	3	7	14	140	7
3	8	4	13	35	455	13
4	10	5	21	82	455	42
5	12	6	31	187	1,558	62
6	14	7	43	422	13,284	86
7	16	8	57	947	23,675	114
8	18	9	73	2,114	59,192	219
9	20	10	91	4,691	145,421	273
10	22	11	111	10,342	351,628	333
11	24	12	133	22,651	838,087	399

2.2.7.3 An Example. We produce LR and LL parse tables for a toy grammar borrowed from [Tomita, 1987]:

(1) S $\rightarrow$ NP VP
(2) S $\rightarrow$ S PP
(3) NP $\rightarrow$ n
(4) NP $\rightarrow$ det n
(5) NP $\rightarrow$ NP PP
(6) PP $\rightarrow$ prep NP
(7) VP $\rightarrow$ v NP

2.2.7.4 LR Table Construction. We need to add an additional production number (0) to simplify LR table generation. The final grammar is:

(0) accept → S $end
(1) S → NP VP
(2) S → S PP
(3) NP → n
(4) NP → det n
(5) NP → NP PP
(6) PP → prep NP
(7) VP → v NP

Table 2.3 shows the steps for generating the LR action table.

Table 2.4 is the actual LR parse table for our toy grammar. The *shift* action is abbreviated to 'sh' and the *reduce* action is abbreviated to 're'. Therefore, 'sh3' is for shift and go to state 3. Note that states 11 and 12 have conflicting actions when the next input's part of speech is a 'prep'. On the one hand, we must shift, and, on the other hand, we have a reduce action.

2.2.7.5 LL Table Construction. For LL(1), we must first calculate the First and Follow sets. Then we can calculate the choice set values using the algorithm in Section 2.2.7.1 for LL(1). Table 2.5 shows the first sets, Table 2.6 shows the follow sets, and Table 2.7 shows the choice set values for our toy grammar.

The LL parse table is shown in Table 2.8. There are seven entries in the table. Thus, we need an array of seven bytes to hold the LL table. This agrees with formula (1), for seven productions and four terminal symbols:

$$7 * (4 /8) = 7 \text{ bytes}$$

The LR table with thirteen states (refer to Table 2.3), four terminal symbols and four nonterminals has the size:

$$13 \times (4 + 4 + 1) = 117$$

We can see how far the LL(1) and the LR(1) space usage varies, even for a little toy grammar.

Table 2.3 LR Table Generation

State	Status of Productions	Action to be taken
0	\$accept $\rightarrow$ $_\wedge$S \$end	S goto 1
	S $\rightarrow$ $_\wedge$NP VP	NP goto 2
	S $\rightarrow$ $_\wedge$S PP	
	NP $\rightarrow$ $_\wedge$n	n shift 4
	NP $\rightarrow$ $_\wedge$det n	det shift 3
	NP $\rightarrow$ $_\wedge$NP PP	
1	\$accept $\rightarrow$ S $_\wedge$\$end	\$end accept
	S $\rightarrow$ S $_\wedge$PP	PP goto 5
	PP $\rightarrow$ $_\wedge$prep NP	prep shift 6
2	S $\rightarrow$ NP $_\wedge$VP	VP goto 8
	NP $\rightarrow$ NP $_\wedge$PP	PP goto 9
	PP $\rightarrow$ $_\wedge$prep NP	prep shift 6
	VP $\rightarrow$ $_\wedge$v NP	v shift 7
3	NP $\rightarrow$ det $_\wedge$n	n shift 10
4	NP $\rightarrow$ n $_\wedge$	. reduce 3
5	S $\rightarrow$ S PP $_\wedge$	. reduce 2
6	PP $\rightarrow$ prep $_\wedge$NP	NP goto 11
	NP $\rightarrow$ $_\wedge$n	n shift 4
	NP $\rightarrow$ $_\wedge$det n	det shift 3
	NP $\rightarrow$ $_\wedge$NP PP	
7	VP $\rightarrow$ v $_\wedge$NP	NP goto 12
	NP $\rightarrow$ $_\wedge$n	n shift 4
	NP $\rightarrow$ $_\wedge$det n	det shift 3
	NP $\rightarrow$ $_\wedge$NP PP	
8	S $\rightarrow$ NP VP $_\wedge$	. reduce 1
9	NP $\rightarrow$ NP PP $_\wedge$	. reduce 5
10	NP $\rightarrow$ det n $_\wedge$	. reduce 4
11	**Conflict: shift/reduce with prep**	
	NP $\rightarrow$ NP $_\wedge$PP	PP goto 9
	PP $\rightarrow$ prep NP $_\wedge$	. reduce 6
	PP $\rightarrow$ $_\wedge$prep NP	prep shift 6
12	**Conflict: shift/reduce with prep**	
	NP $\rightarrow$ NP $_\wedge$PP	PP goto 9
	VP $\rightarrow$ v NP $_\wedge$	. reduce 7
	PP $\rightarrow$ $_\wedge$prep NP	prep shift 6

Table 2.4 LR Table for Toy Grammar

state	det	n	v	prep	$	NP	PP	VP	S
0	sh3	sh4				2			1
1				sh6	accept		5		
2			sh7	sh6			9	8	
3		sh10							
4			re3	re3	re3				
5				re2	re2				
6	sh3	sh4				11			
7	sh3	sh4				12			
8				re1	re1				
9			re5	re5	re5				
10			re4	re4	re4				
11			re6	**sh6/re6**	re6		9		
12				**sh6/re7**	re7		9		

Table 2.5 First Sets

Symbol	det	n	v	prep
NP	X	X		
PP				X
VP			X	
S	X	X		

Table 2.6 Follow Sets

symbol	det	n	v	prep	$
NP			X	X	X
PP			X	X	X
VP				X	X
S				X	X

Table 2.7 LL Table Generation for Toy Grammar

No.	Production	v	prep	det	n	Choice Set Value
1	S $\rightarrow$ NP VP	0	0	1	1	3
2	S $\rightarrow$ S PP	0	0	1	1	3
3	NP $\rightarrow$ n	0	0	0	1	1
4	NP $\rightarrow$ det n	0	0	1	0	2
5	NP $\rightarrow$ NP PP	0	0	1	1	3
6	PP $\rightarrow$ prep NP	0	1	0	0	4
7	VP $\rightarrow$ v NP	1	0	0	0	8

Table 2.8 LL Table for Toy Grammar

Production Number	Choice Set Value
1	3
2	3
3	1
4	2
5	3
6	4
7	8

2.2.8 LL(1) or LALR(1), That is the Question. The two dominant parsing techniques in compilers are LL(1) and LALR(1). Each has its own particular strengths. Fischer and LeBlanc [1988, p. 197], in a section with the same title as this one, compare the two methods in the following aspects:

1. Simplicity: Both have simple drivers. LL(1)'s underlying concepts are very easy to visualize and understand. LALR(1) is rather more complicated.
2. Generality: Although neither method can handle all unambiguous grammars, LALR(1) has a definite edge in generality. All LL(1) grammars are LR(1) and virtually all are LALR(1). LL(1) parsers are fairly strict about grammar forms, forbidding left recursion and productions that share a common prefix.
3. Action Symbols: The interface between the parser and the semantic rules are action symbols. LL(1) allows placement of action symbols anywhere in the right hand side. LALR(1) allows placement of action symbols at the extreme right end of the production, but, may not allow placement

elsewhere. It is worth mentioning here that an LALR(1) grammar can be rewritten to allow necessary semantic routine calls. YACC, an LALR(1) parser generator [Johnson, 1978], allows the placement of code fragments anywhere in a right hand side. Overall, LL(1) allows the most flexibility in action symbol placement.

4. Error Repair: LL(1) parse stacks contain symbols that are predicted, but not yet matched. These symbols provide valuable information in determining possible repairs. The LALR(1) parse stack contains information on what has already been seen. Deciding possible continuations for repair are not as trivial. LL(1) error repair tends to be simpler.
5. Table Sizes: Table sizes in LALR(1) are larger than LL(1). In the worst case, the number of states is exponential in the size of the grammar, see Table 2.2 on page 18. Suffice it to say that because of LALR(1)'s possible exponential blowup in table size, LL(1) has a much safer worst case behavior.

Fischer and LeBlanc [1988, p. 200] conclude that LL(1) has the advantage in all areas except generality, where LALR(1) has a decided edge. Neither method can handle English grammar because of ambiguity. Both of the parsing methods need to be extended to process natural language. Therefore, LALR loses its only advantage in handling larger grammars when it comes to English grammar.

2.3 Spelling Correction

Spelling correction is an important task in an integrated environment. A system cannot simply quit because of spelling errors, since they happen very often. There are three types of parsing approaches in regards to misspelled words:

1. Reject the input completely as soon as a word is considered to have spelling errors. This is easy to implement, but not appropriate in natural language environments, as the user often inputs misspelled words.
2. Inform the user as the first misspelled word is entered, and have him/her correct it. This method is somewhat better than the uncooperative parsing approach above, but not flexible. This approach is used in Tomita's parser [Tomita, 1987].
3. Build a parser that tolerates spelling errors and seldom rejects input sentences because of spelling error. The misspelled word is corrected by a spelling corrector and treated as a correctly spelt word. This approach is often called robust parsing [Tomita 1986, p. 99].

Since one misspelled word may be corrected into many different words, this creates some sort of ambiguity for the parser. The underlying parser must be prepared to deal with numerous replacement words for a single erroneous word. Tomita [1986 p. 99], in an analysis of different methods of parsing in regards to spelling errors, concludes that to deal with serious effects of spelling errors on the parser, the system must have some really complex algorithm, and nevertheless, there is no guarantee that the system can correct all such errors.

The spelling correction process in an integrated parsing environment is not like that in interactive systems, such as word processors, that submit a few candidate words and ask the user to select one. It must provide an automatic correction mechanism and the response should be fast. It is understood that automatic spelling correction is much harder than interactive correction. Obviously, the system cannot simply have a lucky guess and select one of the candidates. It must use contextual information to narrow the choices of the candidates to one. The challenge in automatic spelling correction is to do the job in real time and with high accuracy, without soliciting feedback from the user.

The first task in spelling correction is to decide what words of the lexicon should be compared with the misspelled word. For a small lexicon, we could simply choose all the words in the lexicon to be compared with the erroneous word. For a large lexicon, some type of selection must be in place, or at least, the comparison of an unrelated word with the word in question should be terminated once a limit is reached. Once a word is selected to be compared with the erroneous word, some sort of metric should be considered for finding out the minimum number of changes to the misspelled word to transform it to the word that it is being compared with.

Since there will be more than one word from the lexicon to be compared with the erroneous word, the comparison result should be ranked, so other modules can distinguish amongst the candidate words. There are two extreme conditions that we must watch for:

1. Irrelevant words: Some or most of the words from the lexicon that are compared with the word in question are irrelevant. There should be a mechanism that triggers this fact in the early part of the comparisons. Suppose we are comparing the word ‘teirs’ with the word ‘parent’, which is an obviously irrelevant word to a human being. After the first three characters mismatch, there is no reason to waste time; we should conclude that these two words are unrelated.
2. Close candidate: On the other extreme suppose the word from the lexicon and the erroneous word are very similar. For instance, if we are comparing the word ‘teir’ with the word ‘their’, we could conclude that ‘their’ is a good substitute for ‘teir’, and terminate the spelling correction process.

We could divide the process of spelling correction in an integrated natural language environment into two basic categories:

1. detection of the error, and
2. automatic correction.

The detection of spelling errors is easy; when the lexical analyzer cannot find a matching word in the lexicon for an input word, the word is considered to be misspelled. This method only detects a misspelled word when it is an isolated word. We will see later that a mistype of a word can result in another word that exists in the lexicon. The larger the lexicon, the bigger chance of having a word garble into another word in the lexicon. In a real word, where the lexicon is a full size dictionary, this is not a straightforward task. We will analyze the Unix Spell program later, since all it does is the detection of errors.

The correction of a misspelled word is much harder than the detection of an error. The correction mechanism includes locating the words that are very close to the word, and, making sure that the replacement word does not violate the syntax and semantics of the sentence. An important issue in the correction is the dynamic nature of the language, such as English. That is, words enter and leave the lexicon. Kukich [1992a, p. 83] mentions that no lexicon will ever be complete since novel words will always arise. Unless the lexicon structure is easily updatable, some even question the wisdom of attempts at automatic correction. One big problem is the morphological productivity of the English language, e.g. most nouns can be verbified.

Before getting to spelling correction, it is beneficial to see how spelling errors occur. I categorize spelling errors into the following types:

1. Typographic errors: In this type of error, the user may well know the correct spelling and the error happens by mistake, for example ‘teh’ for ‘the’ or ‘boqk’ for ‘book’.
2. Phonetic errors: In this case, the user does not know the correct spelling, but substitutes a phonetically correct string, ‘physikal’ for physical, or worse ‘fizykal’, or ‘nacheral’ for ‘natural.’ The fact is, when a person does not know the exact spelling of a word, (s)he resorts to phonetic spelling for the word.
3. Cognitive errors: this type of error is based on a misconception or the lack of knowledge. Examples are: ‘recieve’ for ‘receive.’

The above types are the types of real errors. There are two other types of errors that happen frequently: Word boundary and random abbreviation.

2.3.1 Word Boundary. Word boundary errors play a big role in spelling correction. Word boundaries are defined by space characters between two words, or in general blank spaces, tabs and the carriage return [Kukich, 1992b, p. 384]. This type of error happens when either the user thinks there are two separate words or the space character is added or missed by mistake. These two types are known as join and separation. For example,

“upto” → *“up to”*
“specifi cation” → *“specification”*

In the first one we are missing a blank space and in the second case we have an extra blank space. The above examples are relatively easy to deal with. There are complications when the addition/loss of a white space could generate one or two words that exist in the lexicon:

“output” → *“out put”*
“for got” → *“forgot”*
“under stand” → *“understand”*
“in form” → *“inform”*

A simpler kind is when a word is split into two words and the first word is a valid word in the lexicon, but the second part is an isolated word:

“specific ation” → *“specification”*

In this example, the word ‘specific’ is a valid word and we realize we have a misspelled word when we get to ‘ation’.

Another complication occurs when the white space is splitting a word instead of terminating it as in: ‘th ebook’ for ‘the book’. Finally, a white space could be replaced by a regular letter as in: ‘thehmovie’, for ‘the movie’.

The word boundary problem does not stop with two words, for instance “yesthisis”. As Kukich [1992a, p. 84] describes: “The difficulty with run-ons and splits is that most spelling correction techniques rely on the use of white space characters as word boundary makers. Weakening that constraint results in a combinatorial explosion of the number of possible word combinations or subdivisions that must be considered.” Lee [1990] points out that the word boundary problem can also be a result of misunderstanding of technical prefixes, e.g., *cardio*, *milli*.

Kukich [1992b, p. 385] concludes, “errors due to word boundary infractions remain a significant unsolved problem for spelling error detection and correction.”

2.3.2 Abbreviation. Abbreviation of long words is a common method of shortening. However, if the abbreviation is not in the lexicon it is detected as an error. The point to be noted here is that this type of abbreviation is rather random. After typing a few characters of a long word, users often decide to quit. Sometimes they signal an abbreviation by putting a period at the end, for instance:

“spec.” for “specification”
“Christ.” for “Christmas”
“subs.” for “substitute”

This type of abbreviation, which we could call random abbreviation, as the user at some point in the typing decides (s)he has given enough hints, should not be confused with accepted abbreviations like ‘HR’ for ‘heart rate’, or ‘CV’ for ‘Cardiovascular’, as these types of abbreviations are present in the lexicon.

2.3.3 Spelling Correction Process. There are three steps in spelling correction:

1. Detection of an error:
   - 1.1. the result is an isolated word, i.e. ‘teh’ for ‘the’. Basically, the misspelled word does not exist in the dictionary. This type of detection is easy, as the result cannot be located in the dictionary. The Unix spell program [McIlroy, 1982] is based on this type of detection [1].
   - 1.2. the result is a real word. For example ‘of’ in place of ‘if’, or ‘thought’ in place of ‘taught’. The only way that this type of error can be detected is to consider the context of the sentence. The difficulty in that case also is to make a judgement about whether it is a spelling error, the word is misspelled, or is it a genuine syntax/semantics error. There are examples when even the context cannot help. As an example the word ‘late’ in “I *late* the cake,” does not make sense, and the user most likely meant “I ate the cake”. However, if the user types ‘ate’ instead of ‘bake’, assuming a typing error (for I bake the cake), the context is not helpful. Even humans are confused by this type of

[1] In this form of the detection, one must watch for derived words. For instance, the word ‘logically’ can be derived from ‘logic’.

error anyway.

2. Generation of the closest candidate corrections: simply a set of words that are close to the misspelled word. There must be some criteria for deciding when a word is a close candidate for a misspelled word.
3. Ranking of the candidate corrections: This step is to assign some type of measurement to rank the candidates. We could say error distance is the number of characters that have to be changed, deleted or inserted to go from a misspelled word to a word from the lexicon. Then we could define close candidates as the words that have the minimum distance error, within a limit. We mentioned that in some interactive systems, such as word processing systems, the system requires intervention from the user. Most of these systems present a list of replacement candidates to the user and expect the user either to choose one of the suggested words, or to come up with a new replacement word. In such a system, the ranking of candidate words is omitted.

The Unix Spell program [McIlroy, 1982], which only detects spelling errors, is popular with almost everybody. Spell collects words from the named files, and looks them up in a hashed spelling list. Words that do not appear in the list, or cannot be derived from those that do appear by applying certain inflections, prefixes or suffixes, are displayed on the standard output. A simplistic view of Spell, borrowed from [Bentley, 1985], is shown in Figure 2.3.

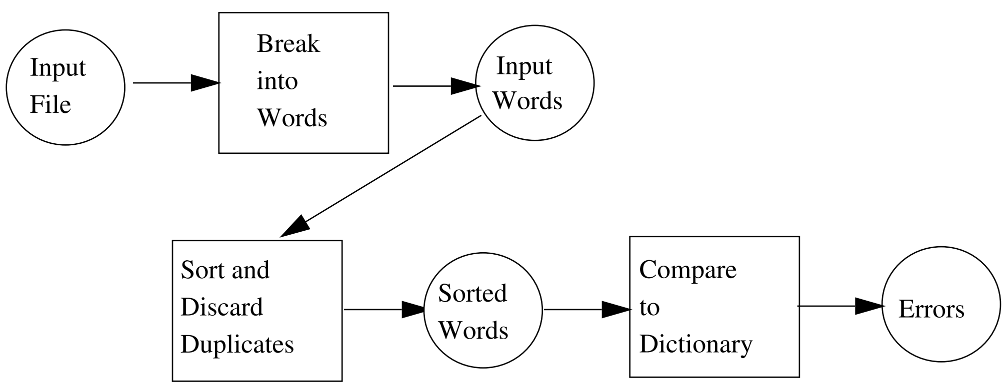


Figure 2.3 A Simple Spelling Checker

It is difficult to arrive at a perfect match for a misspelled word most of the time. The difficulty is that even humans have problems guessing the intended word from among the closest candidates. Consider the word 'ater' [Kukich, 1992b, p. 411] for which we could have the candidates shown in Table 2.9.

In reality, humans use the context for a clue. Examples of this type of usage are:

I saw her, *ater* (**after**) I arrived home.
I saw her *ater* (**later**) on.
Maria did *ater* (**alter**) the clothes.
The drinking ater (**water**) is not good.
She *ater* (**ate**) the Cake.

Table 2.9 Replacement Word for ‘ater’ and the Alteration

Replacement word	Alteration
a**f**ter	Insertion of **f**
later	Insertion of **l**
a**l**ter	Insertion of **l**
water	Insertion of **w**
ate	Dropping the **r**

A spelling corrector cannot select one of these words over the others for replacing ‘ater’, unless it has access to the sentence context.

Damerau [1964] found that approximately 80% of all misspelled words contain a single instance error. Pollock and Zamora [1984] found that only 6% of 50,000 nonword spelling errors were multi-error misspellings. Conversely, Mitton [1987] found that 31% of misspelled words in a 170,016 word corpus contained multiple errors.

Many studies show that few errors occur in the first letter of a word. Pollack and Zamora [1984] found that 3.3% of their 50,000 misspellings involved first letters. Yannakoudakis and Fawthrop [1983] found 1.4% of 568 typing errors were in the first position. Note that words like ‘psychology’ where the ‘p’ is silent are rare but, most likely the first character is omitted in many situations when the error is a phonetic type.

Another source of typing errors is the use of adjacent keys in the keyboard; e.g., the word *possible* is typed as *posdible*. In this case, the character ‘d’ is typed in place of character ‘s’, being next to ‘s’ in the standard keyboard. Most of the errors involving reverse order of characters, like ‘teh’ for ‘the’, happen because of the motor movements of the user. Veronis [1988] refers to this type as performance errors as opposed to competence errors. Performance errors are due to mechanical or neuro-motor problems (typographical errors, slips of the pen), whereas competence errors reflect ignorance about the language rules or misconceptions about the domain.

The spelling error patterns to be considered depend on the application task. Keyboard adjacency problems appear when typewriters or computer keyboards are used; they result from motor coordination errors, such as substitution of the character ‘D’ for ‘S’. These errors are different from problems introduced by an optical character recognizer (OCR). Errors in OCR output are based on confusion, due to featural similarities between letters, e.g. the substitution of D for O. Kukich [1992b, p. 387] describes a number of the spelling error patterns. She found that even two similar text entry modes may exhibit different error frequencies.

An important issue in spelling correction is to find all strings in the lexicon that are close to the misspelled word under some measure of closeness. The most common measure of closeness is known as the edit distance. Wu and Manber [1992, p. 83] define edit distance as “A string P is said to be of distance k to a string Q if we can transform P to be equal to Q with a sequence of k insertions of single characters in (arbitrary places in) P, deletions of single characters in P, or substitutions of characters."

Allowing up to k characters of the pattern to mismatch with the corresponding text reduces the timing spent on irrelevant words from the lexicon. For example, if k=2 the pattern *mismatch* matches *miscatch* and *dispatch*, but, not *respatch* [Baeze-Yates and Gonnet, 1992, p. 76]. Once the limit is reached, the remaining characters of the two words do not need to be compared with each other.

2.3.4 Storage Considerations. Because of the large size of the lexicon, efficient access time is an important issue. The way the lexicon is stored dictates the access time. The dynamic nature of natural language is also an important consideration and the system must be extendable as Kukich [1992a, p. 83] mentions, no lexicon will ever be complete since novel words will always arise. Finally, there should not be any restriction on the number of words in the lexicon.

Two methods of storage stand out amongst the other methods. The first, and the most common technique makes use of a hash table [Knuth, 1973]. The second method for search is a Trie, which was used in the first version of Circsim-Tutor [Lee, 1990].

In hash table techniques, to look up a string in the lexicon, its hash index is calculated. Using the index, the lexicon is accessed. The system must be able to resolve collisions, as two or more words may have the same hash index. The most common form of resolution is to keep a linked list of strings in which their hash index collides. Fischer and LeBlanc [1988, p. 257] put the following requirement on the hash function 'h':

- h (n) depends solely on n.
- h can be computed quickly.
- h is uniform and randomizing in mapping names to hash addresses. That is, all hash addresses are mapped to with equal probability, and similar names do not cluster to the same hash address.

The performance of the system will depend on the hash function. Cormen et al. [1994, p.226] state that a good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots, where m is the size of the hash table. Although it is ideal to have a collision free hash table, in real life it may not be practical. What is important is the time that one submits a string, and the time that the lexical entry of the string is accessed. In other words, if it takes t_h units of time to construct the hash index for string 's', and t_s units of time to get the result from the lexicon, the operation takes: $t = t_h + t_s$ units of time. Constructing a time consuming perfect hash index may not be practical unless it takes less than t units of time. That is, with a perfect hash t_s will be lower as there is no collision, but the question is the increase in t_h. If the hash index creates a collision free index for over 80% of the words in the lexicon, and just a few collisions for the rest, it may be acceptable.

An important consideration here is to note that the usage of words varies. There are words that are accessed much more frequently than others. The hash function must be tuned to generate fewer collisions for frequently used words. To a much lesser extent, when n words hash to the same slot, it is better when they differ in the initial characters rather than the last few characters as the comparison operation fails on the first character mismatch. That is, it is beneficial to fail on the m^{th} character rather than on $(m+1)^{st}$.

In summary, calculation of the hash function must be done in real time. Another constraint is that the size of the hash table cannot be huge.

In the second version of Circsim-Tutor, the lexicon was organized in a trie structure [Lee, 1990]. Lee uses a three level trie. His approach gets us, very fast, to the collection of words that have the same three initial characters as the word being searched for. It is obviously fast for words with length up to three, as we have them directly in the search path. Considering that, on the average, in a large lexicon, most words are longer than three characters we are faced with another search problem: what to do with the list of words with the same n initial characters? For instance, all words in the Unix dictionary with the initial three characters: 'the' are shown in Table 2.10.

Table 2.10 Words with Initial Characters 'the'

the	Thea	theatric	Thebes	thee
theft	their	theism	theist	Thelma
them	thematic	theme	themselves	then
thence	thenceforth	theocracy	Theodore	Theodosian
theologian	theology	theorem	theoretic	theoretician
theorist	theory	therapeutic	therapist	therapy
there	thereabouts	thereafter	thereat	thereby
there'd	therefor	therefore	therefrom	therein
there'll	thereof	thereon	Theresa	thereto
theretofore	thereunder	thereupon	therewith	thermal
thermionic	thermistor	thermo	Thermofax	thermostat
thesaurus	these	theses	Theseus	thesis
thespian	theta	Thetis	they	they'd
they'll	they're	they've		

Once we get to the collection of words with the same three initial characters, other methods of search must be used to locate the word in the list. Obviously any list in the Trie is smaller than the lexicon, but the tail lists could be very long.

To measure the length of words and their distribution, I wrote a program that reports on the size of words in a dictionary. Table 2.11 shows the result of running the program on the Unix dictionary. There are 25,144 words in the dictionary. Column 'Length' displays the number of characters in a word. Column 'Occurrence' shows the number of words that have 'Length' characters. The last column shows the percentage of the words in the dictionary.

Figure 2.5 shows the relationship between Unix Dictionary words and their length.

Table 2.11 Word Length Distribution in the Unix Dictionary

Length	Occurrence	Percentage
1	26	0.103404
2	133	0.528953
3	800	3.18167
4	2194	8.72574
5	3169	12.6034
6	3877	15.4192
7	**4072**	**16.1947**
8	3622	14.405
9	3093	12.3011
10	1973	7.8468
11	1121	4.45832
12	592	2.35444
13	280	1.11359
14	116	0.461343
15	44	0.174992
16	17	0.0676106
17	7	0.0278396
18	4	0.0159084
19	0	0
20	1	0.00397709
21	2	0.00795418
22	1	0.00397709

In this test, the most frequent word length is 7. The average word length is 7.21. Analysis of the word length in the lexicon of Circsim-Tutor version one is summarized in Table 2.11.

The average word length is 6.78 characters. There is a total of 1567 words. 1492 words have more than three characters. So using a Trie of degree 3, causes these words to be in the last partitions. Even if the partitions are equally distributed, there will be 56 (1492 / 26) words on each list. Of course, some lists have a much higher number of words than others. Note that 56 is large enough to prohibit linear search. Even if we use binary search, we will have:

$$log_2(57)/2 = 3$$

Table 2.12 Word Length Distribution in the Circsim-Tutor Lexicon

Length	Occurrence	Percentage
1	22	1.40396
2	53	3.38226
3	90	5.74346
4	184	11.7422
5	196	12.508
6	204	13.0185
7	**223**	**14.231**
8	194	12.3803
9	133	8.48756
10	114	7.27505
11	79	5.04148
12	32	2.04212
13	25	1.59541
14	10	0.638162
15	7	0.446713
16	1	0.0638162

searches on the average, to find a word in a list. It also takes three comparisons to get to the final list (using the characters to navigate through the Trie). Thus, on the average the Trie structure produces:

$$3 + 3 = 6$$

searches for the appropriate word. Figure 2.5 shows the relationship between Circsim-Tutor words and their length.

Lee [1990] mentions that his trie takes over 18,955 lists, for a lexicon of 1567 words. Each branch node contains 27 linked fields, 26 letters of the alphabet and a blank. At level 1 the branch node contains 26 letters of the alphabet and a special character. At any level all key values are partitioned into 27 disjoint classes depending on their first character. So, there exist a total of ((27 * 27) * 26 + 1 = 18,955) lists. In comparison, the hash table size has only 1949 words as the overhead, see Section 6.10.1 on page 137.

A good hash table organization with a good hash function provides a better and faster search as well as saving a lot of memory compared to a trie, as most of the entries in the trie will be blank.

Usage of hash tables for large lexicons is common. For instance, the Unix Spell program [McIlroy, 1982] uses a hash table.

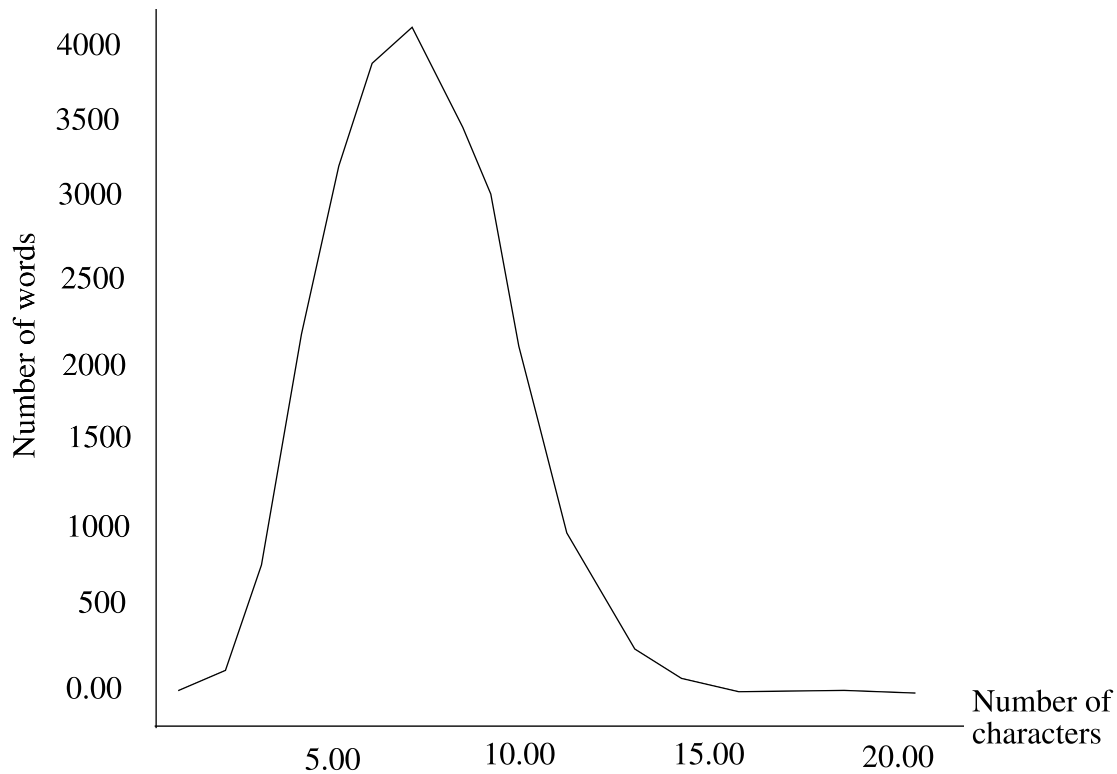


Figure 2.4 Unix Dictionary Word Length

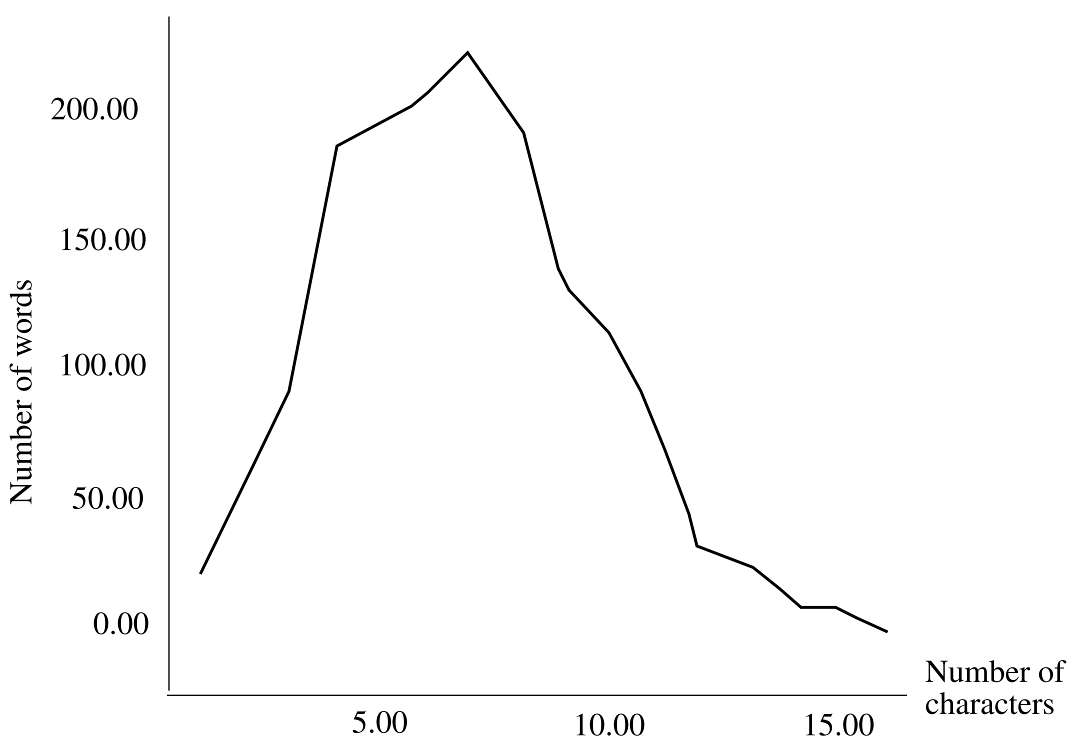


Figure 2.5 Circsim-Tutor Lexicon Word Length

2.3.5 Using Syntax to Improve Spelling Correction. The syntactic and semantic rules of the language are helpful for choosing the better candidate amongst a few possible matching words when there are different parts of speech amongst the candidates. Humans correct spelling errors by using the context of the sentence.

In the NOMAD system [Granger et al., 1983], when an erroneous input word is encountered, the parser gives a list of possible parts of speech that are syntactically acceptable. Then, the lexicon only compares the words in the given category with the erroneous word.

Hall and Dowling [1980, p. 397] use the notation G = (V_N, V_T, P, S) for a grammar, V_N are the nonterminals, and V_T are the terminals, P the productions, and S the start symbol in V_N. In error-correcting parsing, instead of rejecting a string found to be in error during the parsing, the string is corrected to that member of L(G), the language generated by the grammar G, which is closest to the given string. Carter [1992] also has spelling correction using context.

Lee [1990] discusses his Ill-Formed Input Handling System. He uses a three way match algorithm based on the best match algorithm developed by Hall and Dowling [1980]. Carter [1992] discusses a lattice approach to handling word boundary infractions.

Kukich [1992b] reviews the literature on spelling detection and correction techniques. A section is dedicated to natural language for handling dangling ill-formed input (p. 417).

2.3.6 Spelling Correction Summary. In summary, a good spelling corrector must select a small number of words from the lexicon for comparison. Then, the two words (the erroneous word, and the next word from the lexicon) must be compared character by character, in an efficient way. The result of each comparison must be weighted. Finally, the spelling corrector must rank the acceptable candidates so other modules of the system can choose a word from the list of candidates that not only is a good spelling substitute, but satisfies other constraints, i.e., syntax and semantics.

2.4 Lexical Ambiguity

When there is more than one meaning for a single word, we have a lexical ambiguity, e.g., the word ‘saw’ is a noun in ‘chain saw’ and it is a verb in ‘I saw a man’. The parser needs to handle lexical ambiguity as well as syntactic ambiguity. It must be able to handle words with multiple parts of speech.

Most words have several meanings, called *senses* [Allen 1987, p 8]. Often a word has multiple senses with one part of speech. For example, the word *dish* has multiple senses as a noun as well as a sense as a verb.

The biggest problem with lexical processing, and natural language processing in general, is handling the ambiguity that comes up in many forms in natural language processing. One of the important issues in word recognition is determining the word meaning and its part of speech. Not all words have a single meaning and a single part of speech. A good source of examples for lexical ambiguity in English is [Read et al., 1988]. The book is a database of “exemplars” of lexical ambiguity in natural language understanding.

As an example of a word having multiple meaning, lets look at the use of the word “have”, in the following sentences:

1. I have a new car.
2. I have a friend.
3. I have good news.
4. I have a cold.
5. I had breakfast.
6. This shirt has a nice color.

Read et al. [1988, p. 40] states that:

> Words like *have* are highly ambiguous. In addition to the meaning of possessing an object, *have* also indicates many more abstract relationships between the other constituents of the phrase. In the example above, one sense of *have* is that the person is sick; another is that they ate a meal; and still another is that one of the properties of an object has a certain value. It is difficult to define all possible meanings of highly polysemous words like *have*.

The lexical ambiguity applies to the phrases too, i.e., the phrase “ran into” has different meanings in the following sentences:

1. While I was driving home, I **ran into** a parked car.
2. While I was driving home, I remembered I needed some milk. I **ran into** a Seven-Eleven and picked up a half-gallon.

In the first example ‘ran into’ means a vehicle accident took place. In the second it means that someone changed location quickly, [Read et al., 1988, p. 37].

The phrase **‘Out of**’ is highly ambiguous, [Read et al., 1988, p. 48]:

1. I live **out of** town: Location of one object relative to another.
2. I hit her **out of** anger: Relationship between an emotion and an action.
3. I threw the ball **out of** the window: Location an object passed through.
4. The statue is made **out of** marble: Material an object is made of.

Aside from the lexical ambiguity problems, we have to watch for homonyms, homographs and polysemes. Webster [1988] defines these terms as:

Homonyms: one of two or more words spelled and pronounced alike, but different in meaning. The noun *quail* and the verb *quail*.
Homograph: one of two or more words spelled alike, but different in meaning or derivation or pronunciation. The noun conduct and the verb conduct.
Polysemous: marked by multiplicity of meaning.

Webster [1989] defines these terms as:

Homonym: A word like another in sound and spelling, but, different in meaning, as *chase*, to pursue, and *chase*, to ornament metal.
Homograph: A word of the same written form as another, but of different origin and meaning, as homer, ‘home run’ and homer ‘unit of measures.

Polysemy: diversity of meanings.

The following examples are taken from [Chang, 1992]:

1. Homonym:

bear1 'give birth'
bear2 'tolerate'
bear3 'wild animal'

2. Homograph:

lead 'be the leader'
lead 'heavy metal'

3. Polysemy:

bachelor 'unmarried man'
bachelor 'academic degree'
bachelor 'subservient knight', etc.

Miller [1995] mentions that adictionary is an alphabetical list of words. A word that has more than one sense is polysemous; two words that share at least one sense in common are said to be synonymous.

An interesting point in handling lexical ambiguity is expectation [Read et al., 1988, p. 64]. Consider the following two sentences:

1. John asked Mary to marry him. He gave her a ring.
2. John wanted to talk to Mary. He gave her a ring.

In the sentence: *He gave her a ring*, ring could be a phone call or a finger ring. Having expectations associated with *gave* does not resolve this ambiguity. We need to invoke the previous sentence, which provides the context.

It is clear that lexical ambiguity is a serious problem in natural language processing. Our handling of lexical ambiguity is designed to allow a word or a phrase to have multiple meanings. The parser will consider each meaning of a given word one at a time.

When there are multiple choices of part of speech for a single word, the lexical recognizer must provide all of them to the syntactic analyzer and the rest of the system, i.e., the semantic analyzer, must narrow the choice. For instance, the word 'saw' has two parts of speech, verb, and noun. As a verb, it is the past tense of 'to see'. In the following two sentences it is clear which part of speech is to be used:

I saw a man.
I need a chain saw.

My parser is able to parse sentences containing words with multiple parts of speech with several senses for each part of speech. In fact, it is this capability of the parser that helps the spelling corrector to narrow the choices of the candidates.

CHAPTER III

THE PARSER

3.1 Introduction

In the preceding chapter, it was established that a natural language parser must deal with inherently ambiguous grammars. We also saw that there are many words that have multiple parts of speech. And finally, we noted that the inputs are not always in the form of a complete sentence.

Handling ambiguity must be done in an efficient way. Although there are universal parsers, (e.g. Earley's method), that can handle any ambiguity, their performance is not acceptable. This is because in an integrated natural language processing environment, such as Circsim-Tutor, the parser is not the only module of the system. There are also other tasks to be performed, such as text generation. Since this is a real time system, the user expects fast response. The efficiency of the parser is, therefore, critical.

Trying to develop an efficient parser with no restrictions at all, I chose the LL parser because of its speed and its natural top down approach. It was known from the start that the LL algorithm must be extended to handle ambiguity. Aside from the limitations of the technique (no left recursive nonterminal is allowed), there is the problem of reparsing an already parsed section of the input.

3.2 Extended LL(1) Parsing with a Chart

The biggest problem with LL parsers is the restrictions on the grammar: 1) Left recursion is not allowed. 2) Different productions of the same nonterminal are not allowed to have common initial symbols. Although the modified LL(1) algorithm is equipped to handle both deficiencies, my parser generator optionally removes direct left recursion and factors common symbols of nonterminal productions so that the parser can achieve higher speed. The program actually eliminates many types of LL(1) ambiguity in the grammar. Finally, it generates the necessary LL(1) tables.

One important aspect of this parser is that it is table driven. Changes in the grammar do not require any modifications to the parser code. Of course, any grammar change requires construction of a new parse table, which is generated automatically by the parser generator program. In other words, if I decided to parse the Persian language instead of English, I would only need to provide a grammar and a lexicon for the Persian language.

I will not go into the details of an LL(1) parse algorithm or even a chart parser algorithm as they are adequately covered in several textbooks on the subject. An overview was also given in Chapter II. My focus will, instead, be on extension to the LL(1) parser algorithm and the interface between the parser and the chart. However, I will describe the conditions for a grammar to be in LL(1) form.

3.2.1 LL(1) Condition. In this section, we briefly examine the core conditions for a grammar to be LL(1). Let us assume that N is a nonterminal with m productions: $p_1, p_2, \ldots, p_m$:

1. The first set of N, should not contain N. This condition enforces removal of left recursion.
2. The choice sets of all productions $p_1, p_2, \ldots, p_m$ must be disjointed. This condition enforces:
   2.1. There are no common initial symbols in any two productions of N.
   2.2. There is no ambiguity in the grammar. For a given input token, there is at most one production that can be used.

None of the above conditions can be satisfied in a natural language grammar. We will extend the algorithm and combine it with other techniques so there will be no restrictions on the grammar.

3.2.2 Use of Chart. Because of the possibility of multiple parse trees (branches) for a given sentence, many segments of the input sentence are ordinarily parsed and reparsed. To avoid reparsing, a chart, an array of structures is used. The input locations are used as an index to the chart. The input tokens, words, are numbered from 0 up. The last input is followed by the *End Of File* symbol EOF. Each input word is saved in the corresponding location of the chart as its location in input: the first word at location 0, the second one at location 1, etc. As soon as a sequence of words is recognized, the segment, along with its parse tree, is saved in the chart. A segment of input is recognized when it can be reduced to a nonterminal. In other words, a parse tree that has the input word at location ‘n’, as its first leaf node, is saved in the chart at location ‘n’.

We enter the nonterminal symbol N in chart location i as $N(k, r)$, which indicates that N has a parse tree that consumes inputs from location i to $k-1$ using its production number r.

During the process of parsing, we encounter a lot of parsing failure. It is beneficial to record such a failure too. When a nonterminal is rejected, the failure is recorded in the chart, so that we do not have to reparse and fail again.

At any instance, when a parsed sequence is revisited by the parser, the chart is searched for it. The usage of the chart has made it easy to use LL to handle ellipses.

3.2.3 Ambiguity Handling. My extensions to the LL(1) algorithm enable the parser to handle common left symbols between productions of a nonterminal. It can also cope with the ambiguity that arises when more than one production of a nonterminal is a candidate for expansion, given the next input word. This is done simply by allowing the choice sets of a nonterminal productions to have common symbols, and by expanding the nonterminal to all the productions that have the next input token in their choice set.

3.2.3.1 Handling spread of ambiguity. A parser must be able to slow down the spread of ambiguity. Otherwise, the rapid growth of ambiguity will bring the parser to its knees in an ambiguous environment such as natural language.

Each level of the parse tree is automatically isolated from the ambiguity beneath it. When there is more than one parse tree for nonterminal N, all parse trees are saved in the chart at the location of the chart at which N starts parsing the input. Parse trees of N are sorted in ascending order of input consumption. However, for other nonterminals, at higher locations in the chart, that use the parse tree of N, we only record one entry in their parse tree for each different input consumption of N. For instance, let us say that "(N: $i\,j$)" indicates that the nonterminal N has parsed input words from location 'i' up to 'j'. Now, if there is more than one entry of '(N: $i\,j$)' in the chart (indicating that there are more than one way to parse input words from location 'i' up to 'j' in to nonterminal N), all other parse trees that use N, record one such tree. Note that if '(M: $k\,l$)' uses '(N: $i\,j$)', it implies that $k \le i$ and $l \ge j$.

This is what Tomita [1987] refers to as *local ambiguity packing*. In his parser, the program examines the graph-structured stack (see Section 2.2.5 on page 10) to do *local ambiguity packing*. Semantic rules sometimes dictate unpacking of nodes, [Tomita, 1987 p. 43]. Note that we achieved ambiguity packing without much effort. At the same time, we are not losing any information and there is no need for unpacking.

3.2.4 Left Recursion Handling. The problem of left recursion is handled the same way that bottom up parsers do. Expansion of left recursive productions of any nonterminal on top of the parser stack is suspended until the nonterminal has been entered into the chart, using its non-recursive productions. Then, the left recursive productions are expanded and the recursive symbol is simply replaced by the content of the chart.

Handling left recursion is made simple because of the following two steps:

1. Every time the parser reaches the end of a production of a nonterminal, therefore recognizing the nonterminal, it saves the nonterminal symbol into the chart along with the parse tree.
2. Whenever there is a nonterminal on top of the parse stack, the parser first checks the chart to see if the nonterminal has already been parsed for the current input. If it finds the nonterminal in the chart, it simply skips all input words that have already been parsed for the nonterminal. If the nonterminal has more than one parse tree, all of them must be considered one at a time. This allows the parser to maintain multiple parses without parsing any part of the input twice.

As the parser starts with the left recursive nonterminal, it is guaranteed that the first set of its productions are non-left recursive, see Chapter IV for details. Step one, above, guarantees that by the time the parser reaches the left recursive production of a nonterminal, the chart contains at least one parse tree for it. Since there is an instance of the nonterminal in the chart, step 2 above, the parser simply uses the information instead of expanding the left recursive production of the nonterminal symbol, and therefore it skips the problem of handling left recursion common to LL parsers.

It sounds simple and it is simple. There is one catch though; after the left recursive production reaches the end, one or more new parse trees are added to the chart that must be used by the parser, step 2 above, for all the input words that have been consumed by the nonterminal. The solution to this problem is to check all the parse trees of the nonterminal in step 2 one at a time.

The above solution works for direct left recursion. There is complication when we have an indirect left recursion. The system maintains the list of suspended productions. When the first phase of the parsing is done, the system resumes parsing of suspended productions. As an example, the following two productions:

1) $S \rightarrow NP \quad VP$
2) $S \rightarrow S \quad PP$

make 'S' a direct left recursive nonterminal, as it is physically the first symbol the right hand side of the second production.If we have,

1) $S \rightarrow NP \quad VP$
2) $S \rightarrow S_1 \quad PP$
3) $S_1 \rightarrow S$

Then, 'S' is an indirect left recursive. In this case, the system parse the first production of 'S', and while parsing the second production of 'S'', it suspends it. At the end of the parsing, the system resumes parsing of both productions 3, and 2 are resumed.

3.2.4.1 Suspended system. When nonterminal *s* is to be parsed at location 'i', The following are checked:

if it is already parsed (it is in the chart location i) then
    use the parse tree in the chart.
If it is marked rejected, then
    reject the parsing.
If it is in the suspension list of chart location i then
    suspend its parsing (to be resumed later).
Otherwise
    Enter *s* in suspension list of chart location i, and start parsing.

Parsing production 'r' when the top stack symbol 'p' is suspended at location j: we will enter production 'r' at chart location j as suspended for the current part of speech of the input at location j. We may have the same productions more than once in the suspension list, one for each P.O.S. of input at location j.

3.2.4.2 Resumption When the parsing is finished, we enter the resumption phase. We start at location 0 of the chart and reparse each suspended production.

Every time a suspended production is parsed successfully, we only add the subtrees of the nonterminal if the parsed sub-tree is not already in the list of its subtree. The above process is repeated until there is no change in any parse tree.

3.2.5 Parser Interaction with the Lexical Analyzer. The parser can always obtain one look ahead symbol by calling the lexical analyzer. As the parsing progresses, the parser calls the lexical analyzer to get the lexical analysis of the next input word, as it consumes the current input word. If there is no ambiguity, the lexical analyzer attaches one part of speech to the next input word. In case of ambiguity, or when there is more than one candidate word for a misspelled word, the lexical analyzer returns a list of candidates to the parser. The parser processes the list one symbol at a time. It rejects the lexical analysis only if a lexical interpretation violates syntactic or semantic rules.

The lexical analyzer is language dependent, unlike the parser, which is independent of any language. In other words, if we decide to parse Persian instead of English, all lexical entries must be changed. Note that the parser code is generic, but the parse tables are language dependant. The parser generator reads the grammar and creates the parse table.

3.2.6 LFG F-Structure Generation. By using the parser capability of running predefined functions, augmented in the grammar, the parser produces the Lexical Functional Grammar (LFG) f-structure. The algorithm used for f-structure generation is defined in [Wescoat, 1987]. All three well-formedness conditions on f-structure, uniqueness, completeness, and coherence conditions, defined in Section 2.2.6 are enforced.

To show how the LFG f-structure are built, I explain in details a simple grammar rule with augmented Lexical Functional Grammar functional schemata and a lexical entry for a 'verb' with the list of functional schemata. Our simple grammar rule is:

$$\begin{array}{cccc} S & \rightarrow & NP & VP \\ & & (\uparrow \text{SUBJ}) = \downarrow & \uparrow = \downarrow \end{array}$$

Similarly, lexical entries include the LFG functional schemata. Figure 3.1 shows a sample of lexical entry for the transitive sense of the verb "increase".

Representation of item

Syntactic category

increases V ($\uparrow$PRED)= 'increase <($\uparrow$SUBJ) ($\uparrow$OBJ)> '
($\uparrow$SUBJ NUM) = SING
($\uparrow$SUBJ PERS) = 3
($\uparrow$OBJ PAR) = $_c$ +

List of functional schemata

Figure 3.1 Format of LFG Lexical Items

Note that each entry in a f-structure is a pair of the form: (*attribute value)*, e.g. (*NUM SING*). The entries are enclosed in large square brackets in two columns, see Figure 3.2. Attributes are in the first column and the corresponding values are in the second column on the same horizontal line. The *attribute* must always be an atom such as: SUBJ, OBJ, NUM. The *value*, however, can be a simple symbol such as *SING* in (*NUM SING*), or it can be a subordinate f-structure or a semantic form.

After the parse tree is built, all LFG functional schemata, such as the one shown above for the verb *increases*, are copied to the parse tree's leaf nodes. They form the f-structure shown in Figure 3.2 where f_n is the assigned name for the f-structure associated with the node that contains 'increases'.

$$f_n \left[\begin{array}{ll} \text{PRED} & \text{'increase <(} f_n \text{ SUBJ)(} f_n \text{ OBJ)>'} \\ \text{SUBJ} & \left[\begin{array}{ll} \text{NUM} & \text{SING} \\ \text{PERS} & 3 \end{array} \right] \end{array} \right]$$

Figure 3.2 Format of LFG f-structures

Now, the two Lisp functions inserted in the rule propagate the information from the leaf nodes to the top node S. The first function performs the operation: $(\uparrow \text{SUBJ}) = \downarrow$, and the second one performs the operation: $\uparrow = \downarrow$. If f_1 is the f-structure associated with S, f_2 is the f-structure associated with NP, and f_3 is the associated f-structure with VP, the following actions have taken place:

1. If f_1 has an entry with attribute SUBJ, then the first Lisp function equates the corresponding value of attribute SUBJ of f_1, to f-structure f_2.
2. If f_1 is empty or does not have an entry with attribute SUBJ, then the first function creates an entry in f_1 with the following pair: (SUBJ f_2).
3. The second Lisp function equates the two f-structures f_1 and f_3.

Employment of Lexical Functional Grammar concepts, most important, the three well-formedness conditions: *uniqueness, completeness*, and *coherence* conditions, has broadened the parser capability to deal with ill-formed and ambiguous input. The three well-formedness conditions are discussed in Section 2.2.6.1 on page 12. Another benefit is to match the parts of the sentence with the verb arguments (identifying subject, object, etc.). Also, it has enabled the parser to detect ungrammatical input sentences as we saw in Section 2.2.6.3 on page 13 for the sentence: *John falls Mary*.

Lexical Functional Grammar has enabled the context free parser to deal with context sensitive English sentences. This is done by having grammatical requirements for each word of the lexicon. For instance, the word *increase* has multiple parts of speech: 1) *noun*, 2) *verb*. There are two senses for *increase* as a *verb*:

1. Intransitive form for *increase*. In this case there should not be an object in the sentence. Further, the subject must be a parameter. For example:

   Heart rate increases.

2. Transitive form for *increase*. In this case there should be an object in the sentence. Further, the object must be a parameter. For example:

   The baroreceptor reflex increases heart rate.

It is not possible to enforce the above rules (context sensitive as it is particular to the verb *increase*) in the parser. We cannot have specific rules for each sense of a word because we need to deal with all sorts of words combinations. This will simply lead to an unacceptably large number of possibilities. However, using Lexical Functional Grammar we can save all of the above requirements in the lexical entries for the verb *increase*. Figure 3.1 shows the lexical functions for the transitive sense of *increase*. Figure 3.3 shows the lexical functions for the intransitive sense of *increase*.

increases V ($\uparrow$PRED)= 'increase<($\uparrow$SUBJ)> '
($\uparrow$SUBJ NUM) = SING
($\uparrow$SUBJ PERS) = 3
($\uparrow$SUBJ PAR) = $_c$ +

Figure 3.3 Lexical Function for the Intransitive Sense of Increase

Once the parser builds the parse tree of the sentence with *increase* as its main verb, the following actions take place:

1. If there is no object in the sentence, use the intransitive sense of *increase*. Further, we check the lexical function of the *noun* that is the subject of the sentence. If it is not marked 'PAR +', it is not a parameter, so we simply reject the sentence. If however, the subject is of type PAR, the sentence is considered valid. For example, the system will **reject** the input:

   The baroreceptor reflex increases.
2. If there is an object in the sentence, use the transitive sense of *increase*. Further, we check the lexical function of the *noun* that is the object of the sentence. If it is not marked 'PAR +', it is not a parameter, so we simply reject the sentence. If, however, the object is of type PAR, the sentence is considered valid. For example, the system will **reject** the input:

   The heart rate increases the baroreceptor reflex.

The subject of the sentence for both senses of *increase* must be a third person singular.

3.2.7 Handling Words with Multiple Senses. Many words have several parts of speech. The system has lexical entry per one part of speech. Since a word can have several *senses* for a single part of speech, the system allows having separate semantic structures for each *sense* of a given word. For instance, the word *increase* has two parts of speech: *noun*, and *verb*. There are two lexical entries in the lexicon for the word *increase*. Also, the verb *increase* has two senses. The lexical entry for the verb *increase* has a list of two semantic structures, one for each sense of the word. Details of lexical structures are fully covered in Chapter VI.

The parser strictly works with part of speech. The parser allows each lexical entry to have multiple parts of speech. The parser considers all parts of speech of the next input word, and rejects those parts of speech that violate the syntax of the sentence.

Once the parse tree is built, the construction of the f-structure starts. Associated with each lexical entry is its parts of speech and some semantic structures. Included in the semantic structure are LFG annotations. In each parse tree, or the c-structure, a word can have only a single part of speech. Since a word could have several *senses* for a single part of speech, construction of the LFG f-structure considers all senses of the word. The first structure that does not violate the lexical functions is chosen as the result.

To illustrate the concept, we show the process of parsing and f-structure construction for the following sentence:

Increased sympathetic stimulation would ***increase*** *HR.*

Note that the word *increase* has two parts of speech: *noun* and *verb*. The parser excludes the *noun* part of speech of the word because it violates the syntax of the sentence. The verb *increase* has two senses: transitive and intransitive. The f-structure builder excludes the intransitive sense of the verb *increase* because of the existence of the object *HR*.

The fact that the parser strictly works with the input word part of speech and the semantic analyzer deals with input word senses has minimized the effect of lexical ambiguity on the parser. For instance, if a word has n parts of speech and m senses (where $m \geq n$) the parser only deals with n lexical entries rather than m lexical entries. On the average m is bigger than n.

3.2.8 Handling Ellipsis. When a full sentence cannot be recognized, the parser concludes that it has encountered an elliptical sentence. Therefore, the parser tries to match the input with all nonterminals of the grammar. This at first glance might seem to be inefficient but note that most irrelevant nonterminals are eliminated when the first input word is not in their LL(1) choice set, which is a simple look up operation. This is done before the parser actually starts parsing the first word.

All nonterminals that recognize the input string, whether all of it or just a portion, are saved in the chart. The nonterminal(s) that consume all of the input string are considered for the fragment. If there is no recognized nonterminal, no fragment is recognized except in one situation: when the input is a single word. It at least has a part of speech, and that will be our fragment.

For example, the answer to the question "W*hat did you see*?" may be '*a man*' or simply '*man*'. If the grammar has the following rule:

NP $\rightarrow$ noun

then, the input '*man*' is of category *noun* and further it is an *NP*. If the above rule does not exist, then '*man*' is only a *noun*.

3.3 The Parser Algorithm

When I put all the above requirements together and kept in mind the possibility of multiple candidate replacement words for a single misspelled word, I realized from the beginning, that no single algorithm by itself can produce a parser that satisfies the requirements and performs efficiently. The following features produce an efficient parsing algorithm for natural language:

1. Allowing parsing of an ambiguous sentence using top down techniques.
2. The addition of a chart to avoid reparsing.
3. Bottom up mechanism for left recursive productions.

4. Accepting multiple parts of speech for a single word. Within each part of speech, there can be multiple semantic structures, in form of LFG annotations, to allow multiple senses for a single part of speech.
5. Augmentation of the grammar with semantic rules to construct the LFG f-structure, which enforces tougher grammatical constraints.

I made an important design decision to eliminate most limitations associated with the parser. There is no limit to the algorithm as far as:

1. Ambiguity in the grammar.
2. Left recursion.
3. Common left symbols.
4. Number of productions.
5. Number of nonterminal symbols.
6. Number of terminal symbols.
7. No limit on the number of semantic rules in the grammar. Semantic rules can be placed anywhere in a production.

The parser operates as follows:

- When more than one production can be used for a given input, all of them are used.
- When there is more than one good candidate for an erroneous input word, the parser is used to help the spelling correction algorithm to find the best possible correction by using the context of the sentence. The parser determines if any of the candidates violate the syntax and the semantics of the language. Using the context for spelling correction is a very natural method that humans use.
- When handling fragments and ellipses, the system requires no special work. No added productions or new nonterminals are needed.
- When a nonterminal is recognized it is put in the chart.
- When a nonterminal is rejected, the failure is recorded in the chart, so that we do not have to reparse and fail again.

3.4 Example

An example is provided to clarify the concept. The grammar and the lexicon (simple and limited) are borrowed from [Tomita, 1987]. The grammar plus LL(1) information is shown in Table 3.1. The 'choice set' column shows the expected input entry to expand the corresponding production. The 'conflict symbol' column shows the input words that cause the conflict between the associated production and the production number(s) given in 'conflict with prod. no.' column.

Table 3.1 Sample Grammar and Associated Values

prod.	production	choice set	conflict symbol	conflict with prod.
1	S → S PP	*noun det*	*noun det*	2
2	S → NP VP	*noun det*	*noun det*	1

Table 3.1 Sample Grammar and Associated Values

prod.	production	choice set	conflict symbol	conflict with prod.
3	NP $\rightarrow$ *noun*	*noun*	noun	5
4	NP $\rightarrow$ *det noun*	*det*	*det*	5
5	NP $\rightarrow$ NP PP	*noun det*	*noun det*	3, 4
6	PP $\rightarrow$ *prep* NP	*prep*		
7	VP $\rightarrow$ *verb* NP	*verb*	*noun det*	

A very small lexicon of nine words was considered for this example. The words and their parts of speech are shown in Table 3.2.

Table 3.2 Sample Lexicon

Word String	Part of speech
I, man, park, telescope	*noun*
a, the	*determinant*
in, with	*preposition*
saw	*verb*

A nonterminal goes into various states as the parsing progresses. The states along with a graphical representation of a nonterminal status are shown in Figure 3.4.

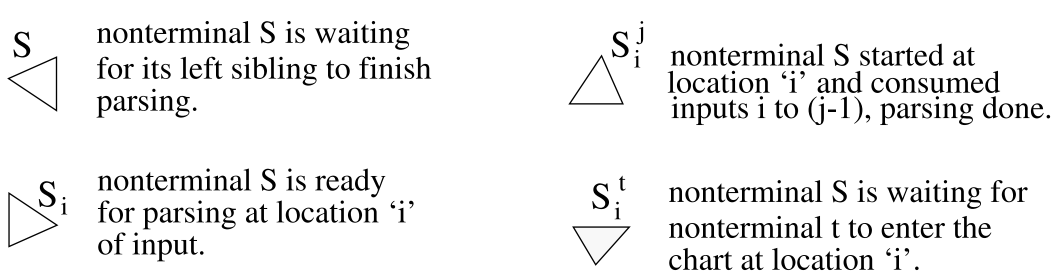


Figure 3.4 Graphical Representation of Nonterminal Status

Two other graphical representations are shown in Figure 3.5.

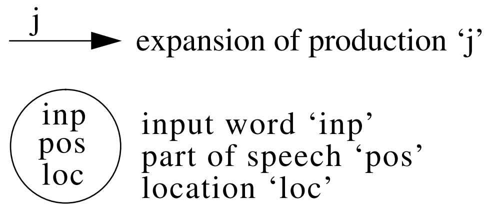


Figure 3.5 Graphical Representation of Nonterminal Status

We parse the infamous sentence:

"I saw a man in the park with a telescope."

We consider that '*I*' is at location zero of input, '*saw*' at location 1 and '*telescope*' at location 9.

Starting the parser we get to *Diagram A*, below, which contains the following information: using the nonterminal symbol S we parse the input starting at location zero. Production 1 and production 2 of S are valid LL(1) choices for this nonterminal. As mentioned above we will use both productions. The first symbol of production one is S, to be parsed at location 0. Since we are currently trying to parse nonterminal *S* at input location 0, therefore in production 1, symbol S needs to wait until *S* is entered into the chart at location 0. In production 1, the nonterminal PP, to the right of *S*, must wait until the parsing of nonterminal *S* has been completed. Production 2 contains two nonterminal symbols *NP* and *VP*. The second nonterminal *VP* must wait until we have finished parsing *NP* and it is entered into the chart.

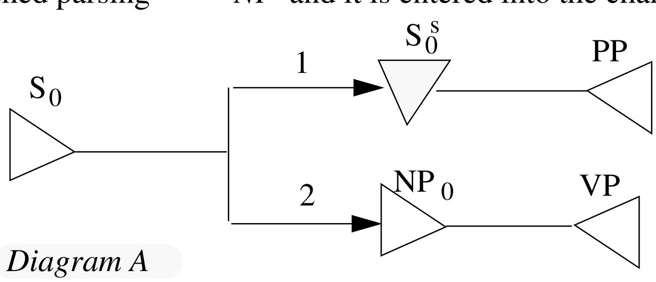


Diagram A

We resume parsing NP at location 0. The LL(1) choice set of *NP* allows expansion of the third production for 'noun'. For the sake of simplicity, we do not consider production 5, since it will fail as the next input is a verb. The terminal symbol 'noun' matches the input 'I,' see *Diagram B*.

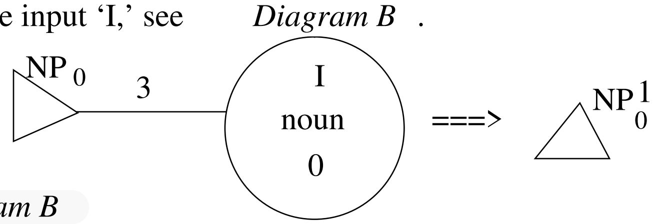


Diagram B

As mentioned in Section 3.2.2 we enter the nonterminal symbol *N* in chart location *i* as *N(k, r)* which indicates that parsing *N* consumes inputs from location *i* to *k-1* using its production number *r*.

NP enters the chart as NP(1,3), consuming input 0 to 1-1 using production 3.

Input #	input symbol	Part of Speech	Recognized Nonterminal
0	I	noun	NP(1, 3)

As NP enters the chart, VP, its right sibling in production 1, is activated and parsing starts at location 1, where NP ended, as shown in *Diagram C1* .

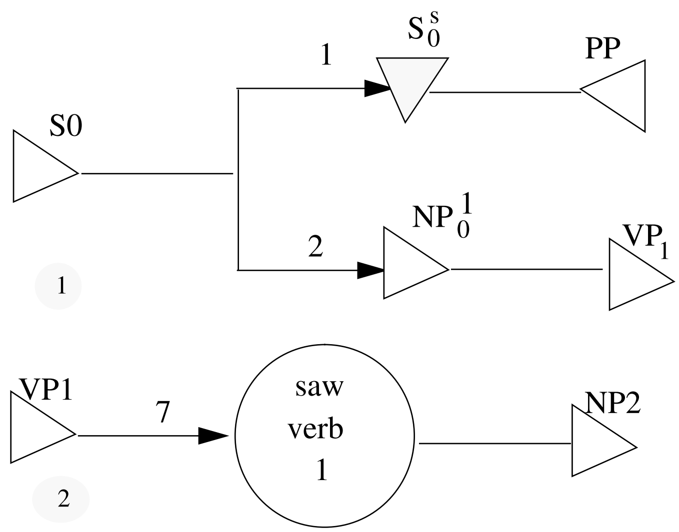


Diagram C

After the expansion of VP using production 7, the terminal symbol ‘verb’ matches the input *saw* , therefore activating its right sibling NP at input location 2 in *Diagram C2* . The state of the parse trees is shown in *Diagram D* .

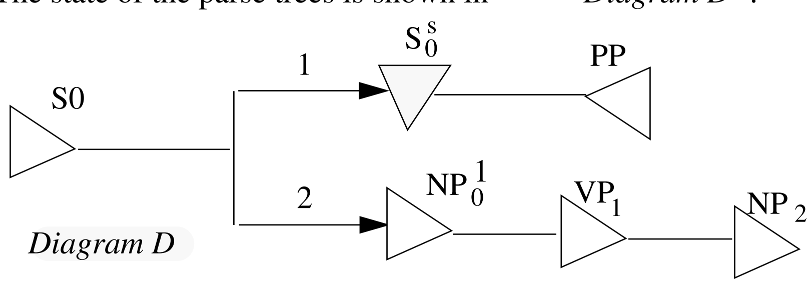


Diagram D

NP resumes parsing at input location 2. Productions 4 and 5 of NP accept input symbol ‘a’ at location 2. The right hand side of the 5th production starts with the nonterminal symbol NP to be parsed with input at location 2. It must wait for NP to be entered into the chart at location 2, since we are currently trying to parse NP at location 2. Production 4 has two terminal symbols that right away match the next two input symbols as shown in *Diagram E* .

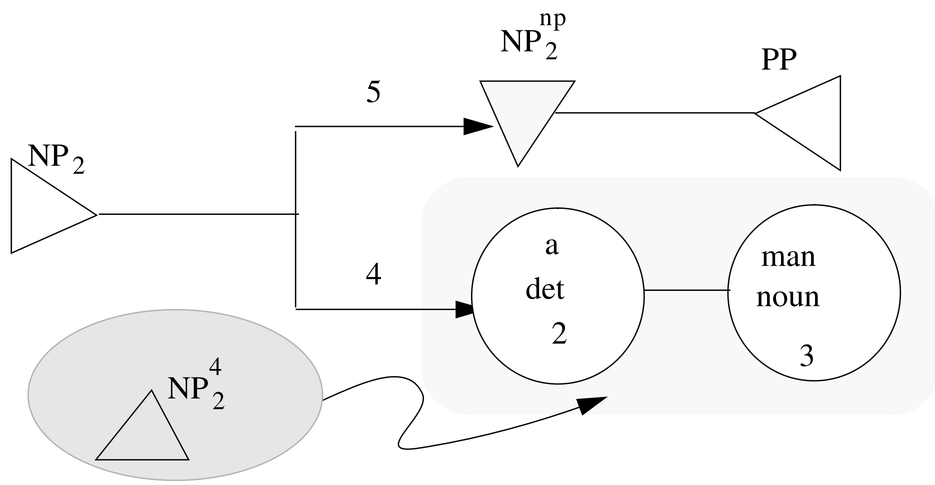


Diagram E

Production 4 terminates and causes insertion of NP into the chart at location 2 as NP(4, 4) since NP has consumed inputs 2 and 3 (up to location 4), using production 4, as shown in *Diagram F*.

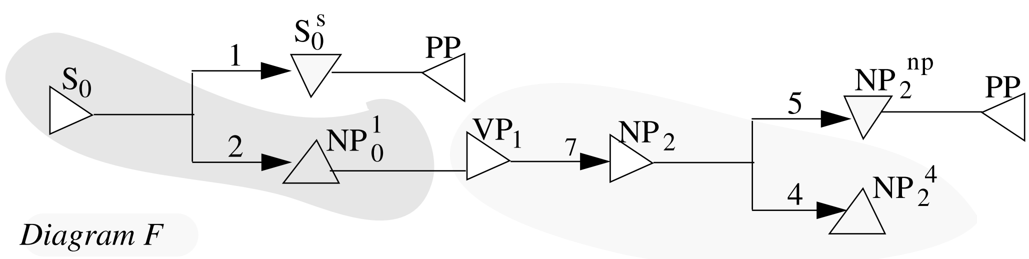


Diagram F

Once NP is entered into the chart, VP_1 is terminated and enters the chart at location 1 as VP(4, 7) since it consumed inputs 1 to 3 and was expanded using production 7 (lightly shaded in *Diagram F*). After VP enters the chart, S_0 is terminated and enters the chart as S(4, 2), dark shaded in *Diagram F*. The chart so far is:

Input #	input symbol	Part of Speech	Recognized Nonterminal
0	I	noun	NP(1,3), S(4,2)
1	saw	verb	VP(4,7)
2	a	det	NP(4,4)
3	man	noun	

We have two recursive symbols S_0 and NP_2 in *Diagram F*. Since we have S at location zero and NP at location 2 of the chart, both of them can be activated. Activation of recursive symbol NP in production 5 is shown in four steps in *Diagram G*. Part 1 of *Diagram G* is copied from *Diagram F*. In the second part of *Diagram G*, we duplicate production 5 and add it to the tree. In the third part we replace the recursive nonterminal with the parsed symbol in the chart. The fourth one shows that the right sibling of the nonterminal is ready for parsing input at location 4.

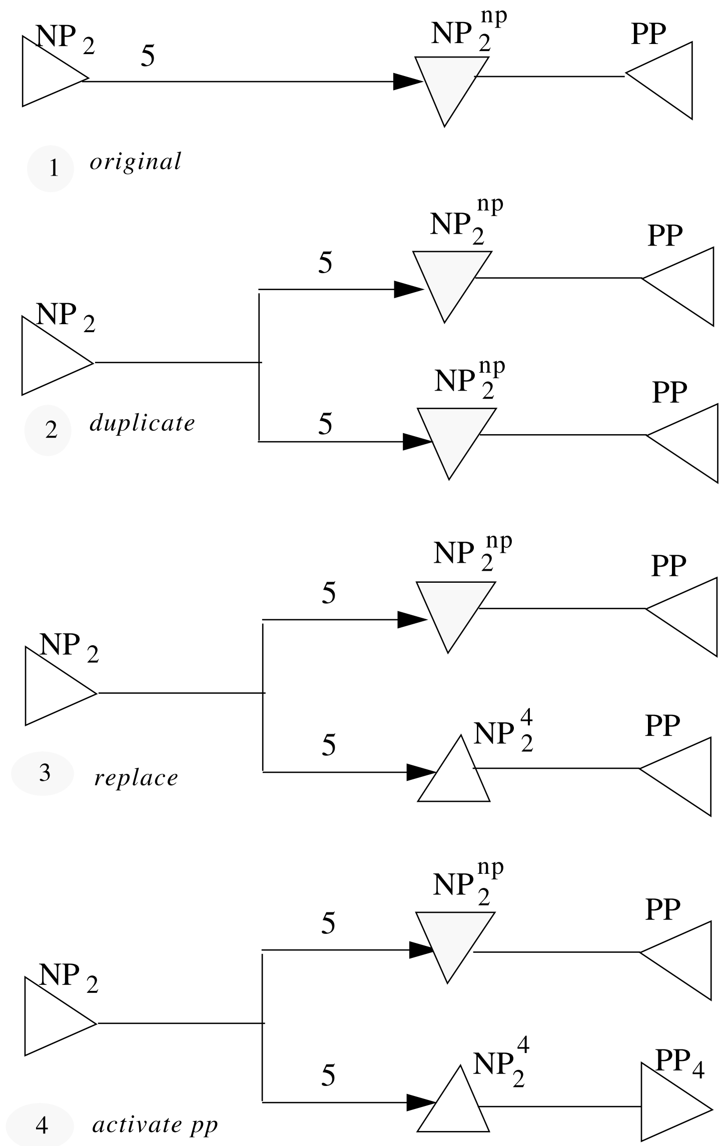


Diagram G

Applying the same procedure to S in production 1 we get *Diagram H* , where shaded areas show the copied productions and the replacement.

PP_4 is the only node in *Diagram H* that can resume parsing. Although there are two nodes marked PP_4, they have been merged so that we do not reparse the second PP_4 node.

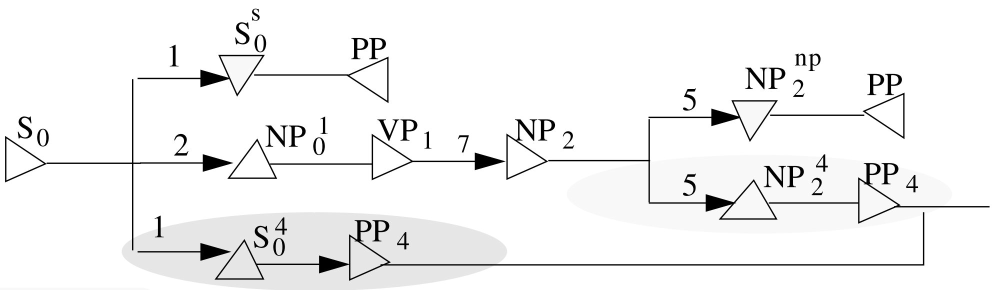


Diagram H

Continuing the process we get to *Diagram I*. *Diagram I* shows expansion of PP's production 6 at input 4. After matching input words at locations 5 and 6 in *Diagram I*, NP's production 4 terminates, causing insertion of NP(7, 4) at location 5 and PP(7, 6) at location 4 of the chart.

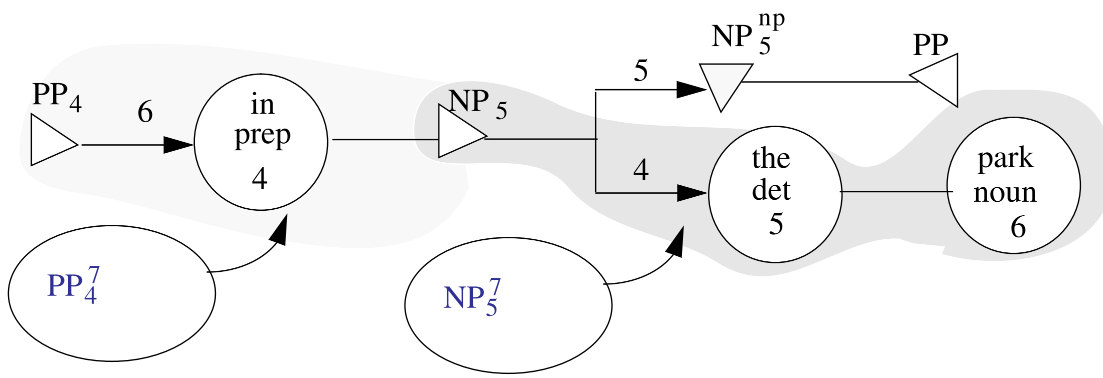


Diagram I

Diagram J shows recursive activation of the nonterminal NP in production 5.

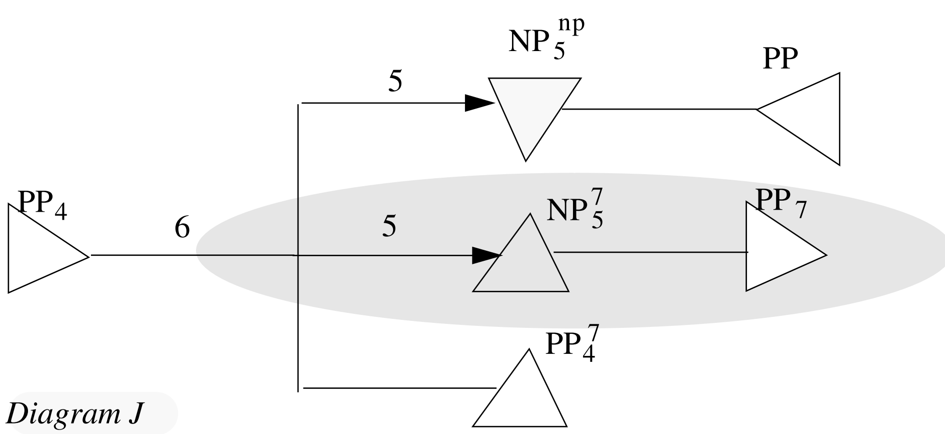


Diagram J

Diagram K shows replacement of PP $_4$ in *Diagram H*. Four shaded areas in *Diagram* K show NP $_2$, VP $_1$ and S $_0$ (in 2 ways) are terminated.

Table 3.3 shows the content of the chart at this point.

PP resumes parsing at location 7 of the input. Two lightly shaded areas in *Diagram*

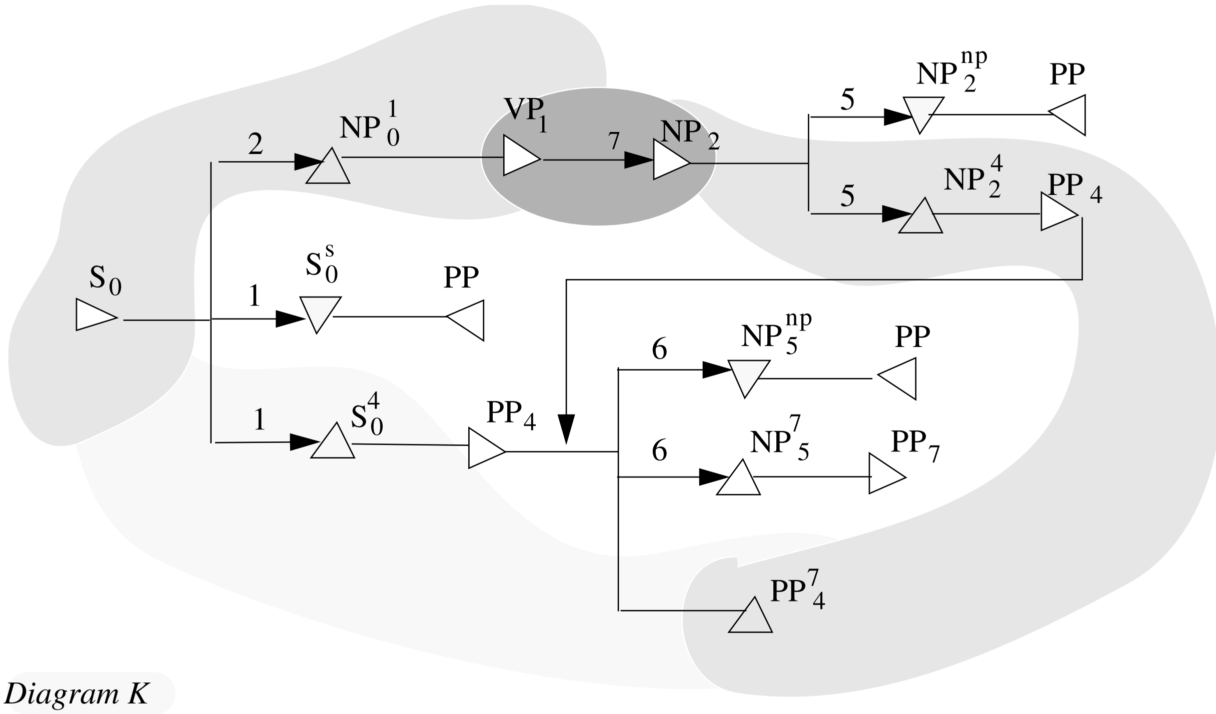


Diagram K

Table 3.3 Chart Contents

Input	Input Symbol	Part of Speech	Recognized Nonterminal
0	I	noun	NP(1, 3), S (4, 2), S(7, 1), S(7, 2)
1	saw	verb	VP(4, 7), VP(7, 7)
2	a	det	NP(4, 4), NP(7, 5)
3	man	noun	
4	in	prep	PP(7, 6)
5	the	det	NP(7, 4)
6	park	noun	

L show the activation of recursive symbols NP_2 and S_0. Also, *Diagram L* shows expansion of PP production 6 at input 7, followed by NP productions 4 and 5 at input 8. The production 4 of NP matches the last two inputs, dark shaded, causing NP(10, 4) to be inserted in the chart at location 8 since it consumed inputs 8 and 9 and was expanded to its 4th production.

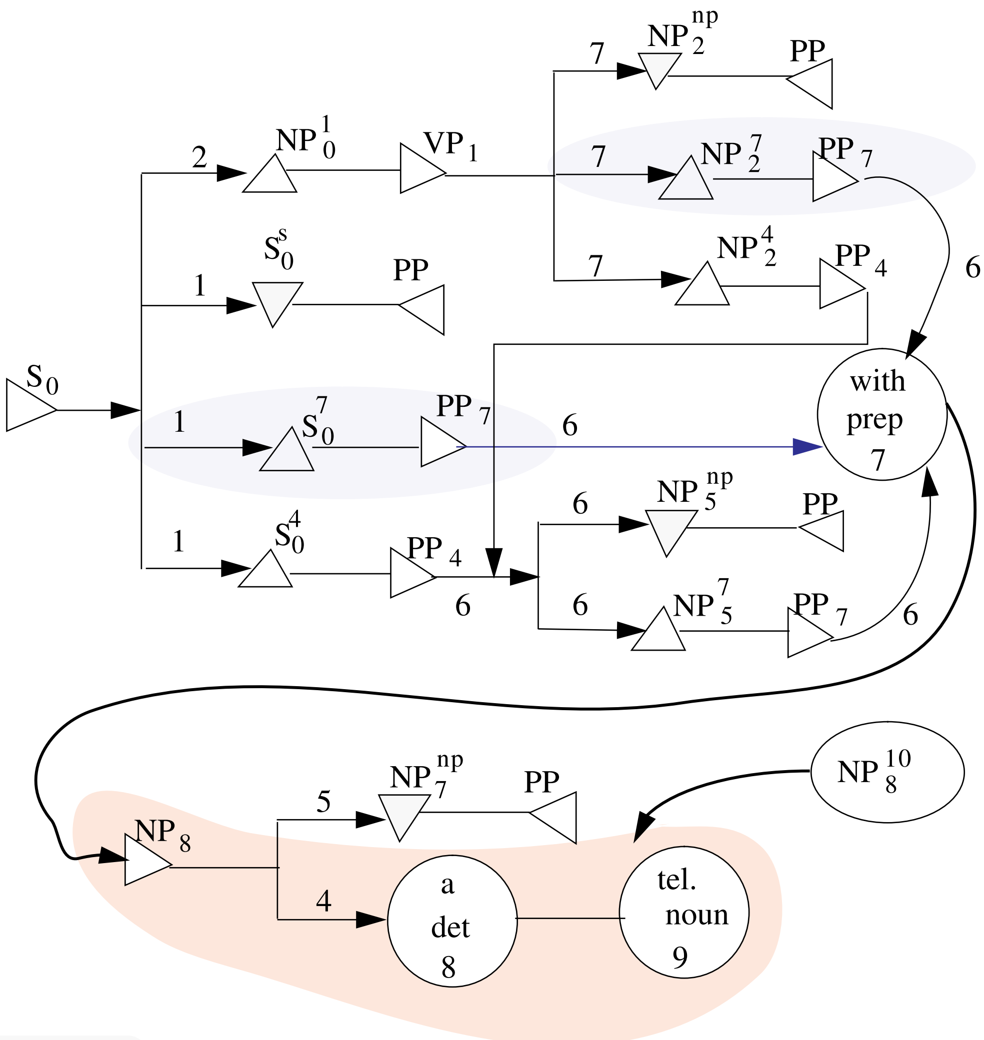


Diagram L

At this point, we have to activate the recursive symbols. The only symbol to the right of the recursive symbols is the nonterminal PP. PP is activated to parse input at location 10. The input at location 10 is the end of file symbol, but PP expects a preposition. Therefore, PP fails. As PP fails, any parse tree that contains PP to parse will fail. The result, after removing all recursive symbols, is the compact forest of parse trees shown in *Diagram M*.

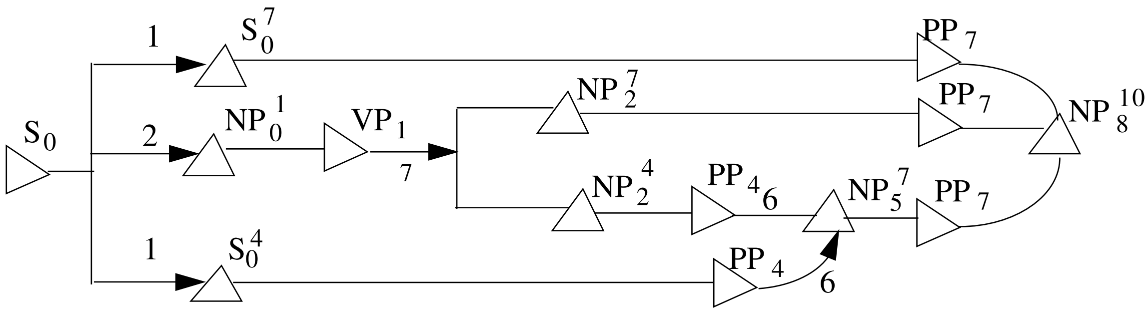


Diagram M

And the final chart is:

Input	Input Symbol	Part of Speech	Recognized Nonterminal
0	I	noun	NP(1, 3), S (4, 2), S(7, 1), S(7, 2), S(10, 1), S(10, 1) S(10, 2)
1	saw	verb	VP(4, 7), VP(7, 7), VP(10, 7)
2	a	det	NP(4, 4), NP(7, 5), NP(10, 5), NP(10, 5)
3	man	noun	
4	in	prep	PP(7, 6), PP(10, 6)
5	the	det	NP(7, 4), NP(10, 5))
6	park	noun	
7	with	prep	PP(10, 6)
8	a	det	NP(10, 4)
9	telescope	noun	

Note that there are repeated entries for NP(10, 5) at location 2 of the chart, and also at location 0 of the chart we have two entries for S(10, 2). This indicates that for the same number of input words, using the same production, the nonterminal has different parse trees. This happens when the production contains one or more nonterminal symbols in its right hand side, and one or more of the nonterminal symbols in the right hand side have more than one parse tree, therefore, the production produces more than one parse tree. For example, in the sentence:

I saw a man in the park with a telescope.

There are two ways to parse " *a man in the park with a telescope* " as an NP. Figure 3.6 shows the parse tree for:

$$NP \longrightarrow (NP: 2\ 7)\ (PP: 7\ 10)$$

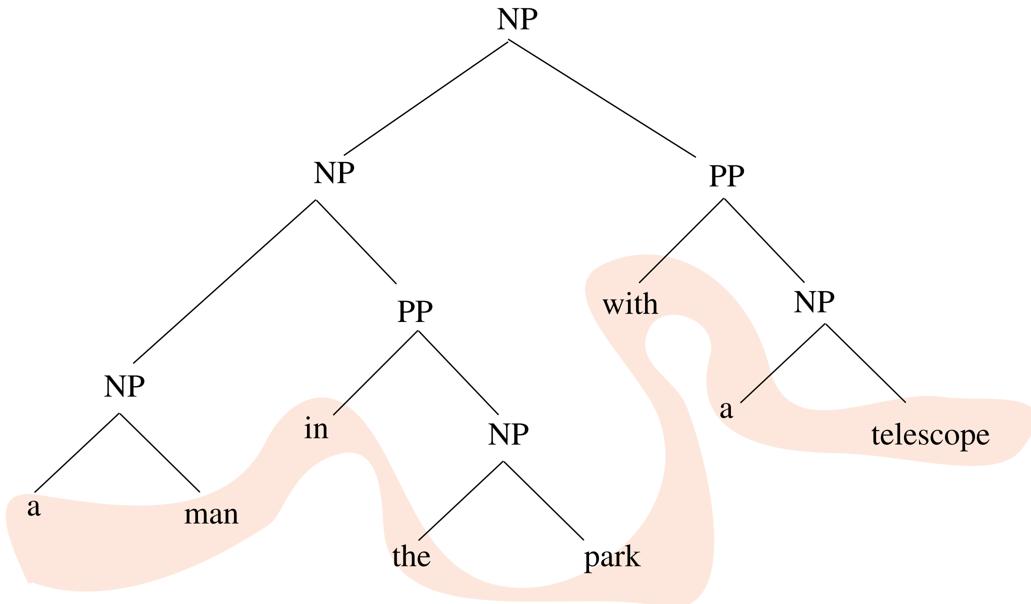


Figure 3.6 Parse Tree for (NP: 2 6) (PP: 7 10)

and Figure 3.6 shows the parse tree for: NP $\rightarrow$ (NP: 2 4) (PP: 4 10)

Now in the production: VP $\rightarrow$ verb NP

we enter VP into the chart as: (verb: 1) (NP: 2 10). This indicates that inputs 1 to 10 are parsed as VP. Note we had a single entry for VP, although there are two ways to parse (NP: 2 10). So, VP is isolated from the multiple parse trees of NP. However, it can access them both.

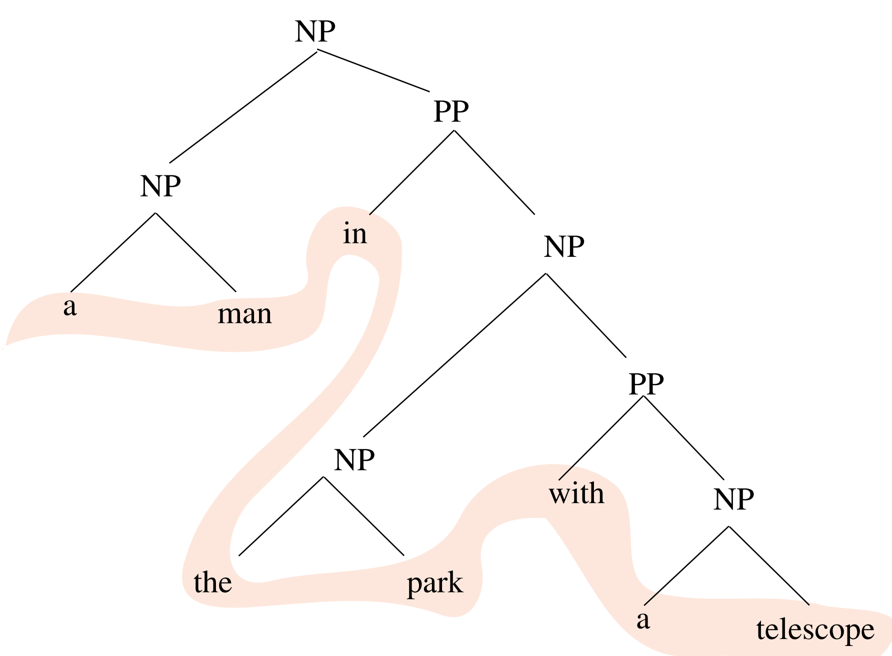


Figure 3.7 Parse tree for (NP: 2 4) (PP: 4 10)

3.5 Test Results

To generate really complicated strings of words that are highly ambiguous, I used the sample sentence borrowed from [Tomita, 1986 p81], the n^{th} sentence is obtained by the following schema:

$$\textit{noun verb det noun (prep det noun)}^{n-1}$$

The fact that the ambiguity multiplies makes the above sentence very difficult to parse for large values of 'n'. Every increment of 'n' causes more difficulty. As a result, the whole forest of the parse trees is affected. But our parser, by using the chart, stores the tree linearly and that is the trick of handling this huge ambiguity even for very large values of *n*.

We analyze, in detail, the parse trees of the above sentence for n=7. The following sentence is generated:

I saw a man with a telescope
with a telescope
with a telescope
with a telescope
with a telescope
with a telescope.

For n=7 we can make up a sentence like:

I saw a man, on the bed in the bedroom, in the apartment, on the main street, at night, in Chicago.

There are 429 parse trees for the above sentence. However, only 120 expanded productions are saved in the chart, saved productions can be combined to produce all 429 parse trees. To see how this is done, the productions of start symbol 'S', as they are saved in the chart, are shown in Table 3.4. The right hand side symbols are shown as (sym: n_1 n_2), which says that the grammar symbol: *sym* was parsed using input at location n_1, and consumed inputs up to location n_2.

Table 3.4 Parse Trees for n=7

No.	Productions	no. of parse tees
1	S → (NP: 0 1) (VP: 1 22)	132
2	S → (S: 0 4) (PP: 4 22)	42
3	S → (S: 0 7) (PP: 7 22)	28
4	S → (S: 0 10) (PP: 10 22)	25
5	**S → (S: 0 13) (PP: 13 22)**	**28**
6	S → (S: 0 16) (PP: 16 22)	42
7	S → (S: 0 19) (PP: 19 22)	132

Note that the reference to the input numbers is with respect to the original sentence that is generated with the following schema:

$$\textit{noun verb det noun (prep det noun)}^{n-1}$$

We investigate the fifth entry in the table further. Figure 3.8 shows a graphical representation of this production. This production actually produces 28 different parse trees. There are 14 ways to parse (S: 0 13) and 2 ways to parse: (PP: 13 22). So, in combination, we have 14 × 2 = 28 parse trees.

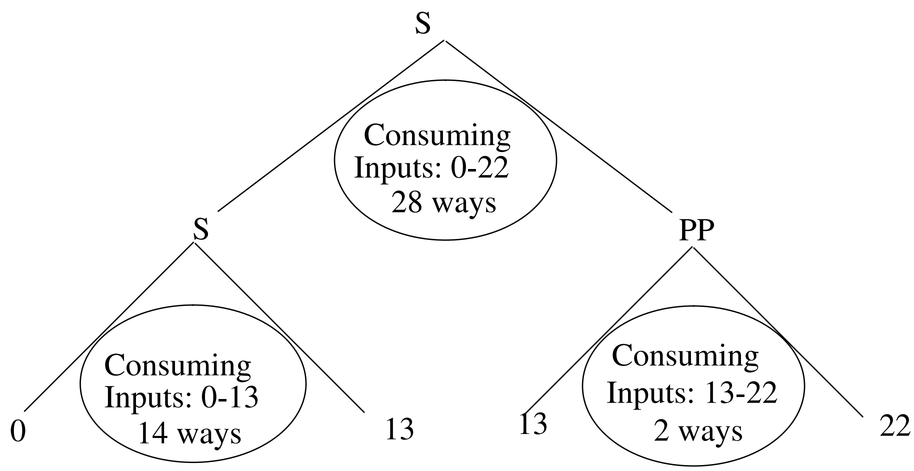


Figure 3.8 Input Consumption of a Sample Production

Here, we briefly describe the parse trees of (S: 0 13) and (PP: 13 22).

1. Two parse trees for S → (S: 0 4) (PP: 4 13). Since there are two parse trees for (PP: 4 13). First, we have PP → (prep 4) (NP: 5 13). And then:
   1.1. NP → (NP: 5 7) (PP: 7 13)
   1.2. NP → (NP: 5 10) (PP: 10 13)
2. Two parse trees for S → (S: 0 7) (PP: 7 13). Since there are two parse trees for (S: 0 7):
   2.1. S → (NP: 0 1) (VP: 1 7))
   2.2. S → (S: 0 4) (PP: 4 7))
3. Five parse trees for S → (S: 0 10) (PP: 10 13). (PP: 10 13) has a single parse tree but (S: 0 10) has five parse trees as follows:
   3.1. Two parse trees for S → (NP: 0 1) (VP: 1 10). As there are two ways to parse (VP: 1 10), since VP → (verb:1) (NP: 2 10). The two parse trees for (NP: 2 10) are:
      3.1. 1. NP → (NP: 2 4) (PP: 4 10)
      3.1. 2. NP → (NP: 2 7) (PP: 7 10)
   3.2. A single parse tree for S → (S: 0 4) (PP: 4 10).
   3.3. Two parse trees for S → (S: 0 7) (PP: 7 10). As we saw before, there are two parse trees for (S: 0 7):
      3.3. 1. S → (NP: 0 1) (VP: 1 7))
      3.3. 2. S → (S: 0 4) (PP: 4 7))
4. Five parse trees for S → (NP: 0 1) (VP: 1 13). There are five parse trees for (VP: 1 13) as VP → (verb: 1) (NP: 2 13) as follows:
   4.1. Two parse trees for NP → (NP: 2 4) (PP: 4 13). Two ways to parse (PP: 4 13) as it expands to PP → (prep:4) (NP: 5 -13)
      4.1. 1. NP → (NP: 5 7) (PP: 7 13))
      4.1. 2. NP → (NP: 5 10) (PP: 10 13)

4.1. 3. NP $\rightarrow$ (NP: 2 7) (PP: 7 13)

4.1. 4. Two parse trees for NP $\rightarrow$ (NP: 2 10) (PP: 10 13). There are two parse trees for (NP: 2 10):

- NP $\rightarrow$ (NP: 2 4) (PP: 4 10)
- NP $\rightarrow$ (NP: 2 7) (PP: 7 10)

Two ways to parse (PP: 13 22) after it is expanded to (prep: 13) (NP: 14 22). (NP: 14 22) generates two parse trees:

1. NP $\rightarrow$ (NP: 14 16) (PP: 16 22)
2. NP $\rightarrow$ (NP: 14 19) (PP: 19 22)

Tomita [1987, p. 33] shows a graphical representation of all 14 parse trees for the sentence:

I saw a man on the bed in the apartment with a telescope

Table 3.5 shows contents of the chart for our sample example with n=7. The chart contains 120 entries. As before, entry *N(k, r)* at location i of the chart indicates that parsing *N* consumes inputs from location *i* to *k-1* using its production number *r*. To show the result in a compact table, Table 3.5 is organized in the following manner:

- Index: index of the chart.
- Recognized Nonterminal: is the nonterminal that is recognized. The production of the nonterminal used for expansion is shown too.
- Input Consumption: The nonterminal symbol parsed input words from location 'index' up to, not including, this number.

The problem gets worse for larger values of n. For instance, for n=10 (having 31 input words numbered 0 to 30), there are 16,796 parse trees. Before getting back to our original task, for testing the behavior of the parser as the sentence grows, we will analyze in details the behavior of the system for n=10.

There are two productions that can parse all input words (from the first word, at location 0, to the last word, at location 30):

(1) S $\rightarrow$ NP VP (2) S $\rightarrow$ S PP

To avoid a long discussion, let us just analyze the behavior of the second production. The second production is more interesting because it combines all the ambiguities of the first production (as it uses 'S' in its right hand side) and contains its own ambiguities too.

In the second production each one of the two right hand side symbols has many different parse trees with different numbers of input word consumptions. The only constraint here is that the first nonterminal (S) must start parsing input from location 0 and finish parsing at location m. Then, the second symbol (PP) must resume parsing input from location m and finish parsing at location *31* (for the end of input symbol). Figure 3.9 shows the compact forest of parse trees using the production:

S $\rightarrow$ S PP

It turns out that there are nine ways that the above production can parse the input sentence, as shown below:

Table 3.5 Productions Saved in the Chart for n= 7

Index	Recognized Nonterminal and its Production	Input Consumption
0	S $\rightarrow$ NP VP	4, 7, 10, 13, 16, 19, 22
0	S $\rightarrow$ S PP	7, 10, 10, 13, 13, 13, 16, 16, 16, 16, 19, 19, 19, 19, 19, 22, 22, 22, 22, 22, 22
0	NP $\rightarrow$ noun	1
1	VP $\rightarrow$ verb NP	4, 7, 10, 13, 16, 19, 22
2	NP $\rightarrow$ det noun	4
2	NP $\rightarrow$ NP PP	7, 10, 10, 13, 13, 13, 16, 16, 16, 16, 19, 19, 19, 19, 19, 22, 22, 22, 22, 22, 22
4	PP $\rightarrow$ prep NP	7, 10, 13, 16, 19, 22
5	NP $\rightarrow$ det noun	7
5	NP $\rightarrow$ NP PP	10, 13, 13, 16, 16, 16, 19, 19, 19, 19, 22, 22, 22, 22, 22
7	PP $\rightarrow$ prep NP	10, 13, 16, 19, 22
8	NP $\rightarrow$ det noun	10
8	NP $\rightarrow$ NP PP	13, 16, 16, 19, 19, 19, 22, 22, 22, 22
10	PP $\rightarrow$ prep NP	13, 16, 19, 22
11	NP $\rightarrow$ det noun	13
11	NP $\rightarrow$ NP PP	16, 19, 19, 22, 22, 22
13	PP $\rightarrow$ prep NP	16, 19, 22
14	NP $\rightarrow$ det noun	16
14	NP $\rightarrow$ NP PP	16, 19, 22, 22
16	PP $\rightarrow$ prep NP	19, 22
17	NP $\rightarrow$ det noun	19
17	NP $\rightarrow$ NP PP	22
19	PP $\rightarrow$ prep NP	22
20	NP $\rightarrow$ det noun	22

1. S $\rightarrow$ (S: 0 4) (PP: 4 31)
2. S $\rightarrow$ (S: 0 7) (PP: 7 31)
3. S $\rightarrow$ (S: 0 10) (PP: 10 31)
4. S $\rightarrow$ (S: 0 13) (PP: 13 31)

5. S $\rightarrow$ (S: 0 16) (PP: 16 31)
6. S $\rightarrow$ (S: 0 19) (PP: 19 31)
7. S $\rightarrow$ (S: 0 22) (PP: 22 31)
8. S $\rightarrow$ (S: 0 25) (PP: 25 31)
9. S $\rightarrow$ (S: 0 28) (PP: 28 31)

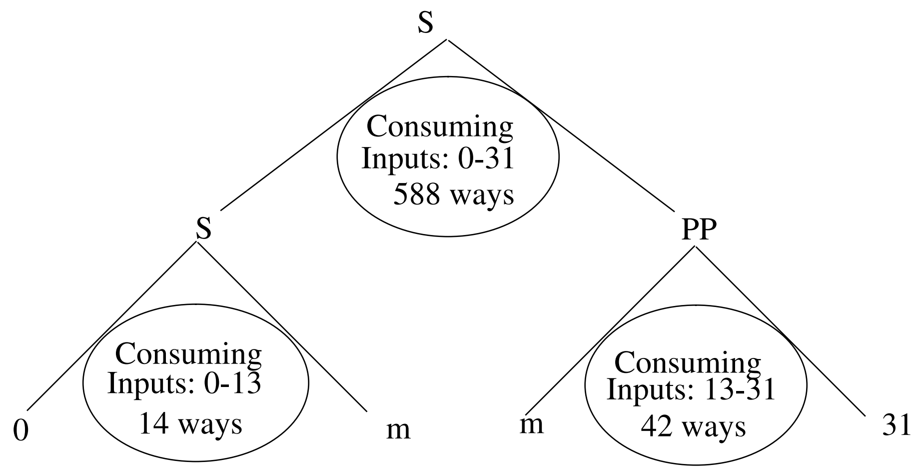


Figure 3.9 Parse Forest for Input Words 0 to 30

Since in the previous example (for n=7), we studied the behavior of (S: 0 13), lets analyze the parse trees of:

$$S \rightarrow (S: 0\ 13)\ (PP: 13\ 31)$$

Here, input words 0 to 13 are recognized as nonterminal S, and input words 13 to 31 are recognized as the nonterminal PP. There are actually 588 parsed trees produced by the above productions. As before, 14 parse trees are produced by parsing inputs starting at location 0 to location 13, as S, and 42 ways to parse input words starting at location 13 to location 31, as PP. As the two segmented parse trees multiply, we get $14 \times 42 = 588$ parse trees.

Actually, for all 16,796 parse trees, only 286 entries are saved in the chart, see the tenth entry in Table 3.6.

Getting back to our original task, for testing the behavior of the parser as the sentence grows, the test was done for ‘n’ starting at one and incrementing it, one at a time, all the way to 50. The snowball effect of ambiguity, in the above example, requires an efficient parser to deal with a large volume of parse trees. Table 3.6 gives a summary of parse statistics for the above sentence for n=1 to n=50. Although the number of ambiguous readings is extremely high (I cannot even read the associated number of ambiguities for n=50), the parser, however, operates efficiently in both time and space.

The first 13 entries in Table 3.6 are also reported in [Tomita, 1986 p. 82]. The headers in the table are:

- N is the number of the propositions (as in the formula).
- Time is the CPU time to parse the sentence in *millisecond* s. Two measurements are presented. The first one is the result of running the

Table 3.6 Lisp Result for n=1 to n=50

n	cpu time Sun	cpu time PC	prod. saved	ambiguity
1	10	55	4	1
2	10	55	10	2
3	10	55	20	5
4	10	55	35	14
5	20	55	56	42
6	20	55	84	132
7	20	55	120	429
8	30	110	165	1430
9	30	110	220	4862
10	40	164	286	16796
11	70	220	364	58786
12	70	275	455	208012
13	90	330	560	742900
14	110	605	680	2674440
15	140	714	816	9694845
16	160	824	969	35357670
17	180	1318	1140	129644790
18	250	1538	1330	477638700
19	290	2197	1540	1767263190
20	360	2637	1771	6564120420
21	420	3241	2024	24466267020
22	520	4229	2300	91482563640
23	630	5108	2600	343059613650
24	720	5767	2925	1289904147324
25	860	6701	3276	4861946401452
26	1050	8678	3654	18367353072152
27	1220	9832	4060	69533550916004
28	1420	12358	4495	263747951750360
29	1670	13896	4960	1002242216651368
30	2020	16313	5456	3814986502092304
31	2280	18949	5984	14544636039226909
32	2800	21146	6545	55534064877048198
33	3000	25320	7140	212336130412243110
34	3440	28616	7770	812944042149730764
35	4060	32461	8436	3116285494907301262
36	4550	36251	9139	11959798385860453492
37	5150	40370	9880	45950804324621742364
38	5970	47895	10660	176733862787006701400
39	6680	52893	11480	680425371729975800390
40	7480	58660	12341	2622127042276492108820
41	9990	68766	13244	10113918591637898134020
42	11550	75907	14190	39044429911904443959240
43	12530	83376	15180	150853479205085351660700
44	13770	93592	16215	583300119592996693088040
45	15310	102765	17296	2257117854077248073253720
46	16620	114134	18424	8740328711533173390046320
47	18190	124680	19600	33868773757191046886429490
48	22010	141652	20825	131327898242169365477991900
49	24420	155602	22100	509552245179617138054608572
50	26270	170048	23426	1978261657756160653623774456

test on a Sun Sparc 5, using the *Carnegie Mellon University Common Lisp*. The second one is result of running the test on a *486 PC* using the *Allegro Common Lisp for Micro Soft Windows*.

- Prod. saved is the number of productions that are saved in the chart.
- Ambiguity: is the number of parse trees.

Further analysis of the test was done by measuring the percentage of the growth for both the number of productions and the number of the parse trees as *n* increments. The result is summarized in Table 3.7. Each entry shows the percentage of growth from the previous value of *n*, and the current value of *n*. For instance, the first entry shows the percentage of growth from *n*=1 to *n*=2. Similarly, the last entry shows the percentage of growth for the number of ambiguities for *n*=49 to *n*=50.

Table 3.7 Percentage of the Growth

n	time growth	prod growth	ambiguity growth	ambiguity/prod
2	100	250	200	0
3	100	200	250	0
4	100	175	280	0
5	200	160	300	0
6	100	150	314	0
7	100	142	325	1
8	150	137	333	3
9	100	133	340	8
10	133	130	345	22
11	175	127	350	58
12	100	125	353	161
13	128	123	357	457
14	122	121	360	1326
15	127	120	362	3933
16	114	118	364	11880
17	112	117	366	36488
18	138	116	368	113723
19	116	115	370	359126
20	124	115	371	1147573
21	116	114	372	3706448
22	123	113	373	12088076
23	121	113	375	39775027
24	114	112	376	131946005
25	119	112	376	440992870
26	122	111	377	14841 10623
27	116	111	378	5026642876
28	116	110	379	17126490373
29	117	110	380	58675851334
30	120	110	380	202064963034
31	112	109	381	699227731321
32	122	109	381	2430587573400
33	107	109	382	8484960256233
34	114	108	382	29738953839249
35	118	108	383	104626002850673
36	112	108	383	36940321 1819262
37	113	108	384	1308655037297346
38	115	107	384	4650891 125973860

Table 3.7 Percentage of the Growth

n	time growth	prod growth	ambiguity growth	ambiguity/prod
39	111	107	385	16579161612289559
40	111	107	385	59270502763935174
41	133	107	385	212472817622274702
42	115	107	386	763660419181357455
43	108	106	386	27515454483371701 16
44	109	106	386	9937646851454898001
45	111	106	386	35972872006968652056
46	108	106	387	130499413394845517648
47	109	106	387	474399083344180058078
48	121	106	387	1727998661081 175861552
49	110	106	388	6306261620272238438318
50	107	106	388	23056662677810730228715

Note that as mentioned, ambiguity is multiplying, but the representation of parse trees in the chart grows linearly. Table 3.7 shows that the percentage of growth for ambiguity is increasing much faster than the percentage of growth of the number of saved productions. The last column shows the ratio of the number of parse trees to the number of saved productions in the chart. The time growth is shown for the Sun workstation only.

For measuring memory usage as well as CPU time, I ran the C version of the parser from n=1 to n=100, on a Sun Sparc machine. Needless to say, the C version is much more efficient than the Lisp version, although the same algorithm is used. It took less than 9 second to parse the generated sentence for n=100. Note that in Lisp, it took about 26 seconds to parse the generated sentence for n = 50, on the same machine. A summary of this test is shown Figure 3.10.

Note that we are not reporting the size of the ambiguity in Figure 3.10, because for n=100 it is too large, and in C, unlike Lisp, we cannot work with such a large number. To report the actual number of parse trees, we need to multiply the number of parse trees of each node of a production. In C, on a Sun machine and PC the hardware word size is 32 bits, effectively the largest number can be 2^{32}. C does not provide a software simulated number. In Lisp, there is a range of integers that are represented more efficiently than others; each such integer is called a *fixnum*, and an integer that is not a fixnum is called *a bignum.* There is no limit on the bignum, as it is supported by software. Although in Lisp, *fixnum* has the same limitation as in C, *bignum* has no boundary. The beauty of Lisp is that the distinction between *bignum* and *fixnum* is hidden from the user [Steele, 1984, p. 14].

A large value of ‘n’ is unrealistic in real life; certainly n=100 is a overkill. The attempt here is to show that the parser is powerful enough to handle serious ambiguities. There is one important issue here, we do not have any lexical ambiguity. In real life, there is lexical ambiguity, as well as syntactic ambiguity. The fact that syntactic ambiguity and lexical ambiguity multiply, justifies the test for large values of *n*. For example, when there are four parts of speech for a word, in the worst case, there will be four different paths in the parse tree. If there are three syntactic ambiguities in each path, we have 12 (3 X 4) parse trees. Other types of

n	Time	Memory	n	Time	Memory
1	16	20	51	1149	244
2	16	20	52	1183	256
3	16	20	53	1249	264
4	16	20	54	1316	276
5	16	20	55	1416	284
[illegible]	[illegible]	24	[illegible]	[illegible]	[illegible]
16	66	[illegible]	[illegible]	[illegible]	396
17	66	48	67	2483	408
18	83	48	68	2583	416
19	83	52	69	2749	428
20	99	56	70	2783	440
21	116	60	71	2866	452
22	[illegible]	64	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	204	[illegible]	7083	804
47	883	212	97	7333	820
48	966	224	98	7566	836
49	1016	232	99	7816	856
50	1083	240	100	8066	872

Figure 3.10 Result of Running the C Version on Sample Sentences

ambiguities could result from:

1. spelling correction, having more than one candidate for a misspelled word. This could get really bad, if each candidate word has multiple parts of speech. For instance, if there are 5 replacement candidates for a misspelled word, and each one of the replacement words has 5 different parts of speech, then, for one word we will have 25 different possibilities. Obviously, this multiplies, when we have more than one spelling error.
2. Grammatical error that causes the parser to insert/delete some tokens, in case error recovery is added to the parser.

All things considered, the parser must be able to deal with a big forest of parse trees. The efficiency includes both time and space as either one of them could create a bottleneck for the system.

3.5.1 Simplifying the Grammar. In an attempt to analyze the effect of ambiguity on the parser performance, lets try to remove ambiguity from our toy grammar, defined in Section 2.2.7.3 on page 18 and repeated here for convenience:

(1) S $\rightarrow$ NP VP

(2) S → S PP
(3) NP → n
(4) NP → det n
(5) NP → NP PP
(6) PP → prep NP
(7) VP → v NP

The main cause of ambiguity is in productions 5 and 6. To a lesser extent, productions 1 and 2 contribute toward the ambiguity. We do three tests as follows:

1. Eliminate PP ambiguity.
2. Eliminate S ambiguity.
3. Combine the above two steps.

3.5.1.1 Eliminate PP ambiguity. Productions 5 and 6 are the main causes of ambiguity. We leave production 5 alone, and simplify production 6 to:
PP → prep det n

Now there is no recursion in PP. It can only consist of a preposition, followed by a determined noun. The result is shown in Figure 3.11.

Count	CPU	prod saved	Trees
1	10	4	1
2	0	9	2
3	10	14	3
4	10	19	4
5	10	24	5
6	10	29	6
46	[illegible]	229	46
47	100	234	47
48	100	239	48
49	100	244	49
50	100	249	50

Figure 3.11 Result with no ‘PP’ Ambiguity

3.5.1.2 Eliminate S ambiguity. To a lesser extent, the two productions:

(1) S → NP VP (2) S → S PP

contribute to ambiguity as there are two ways to recognize ‘S’. We remove the second production to eliminate this ambiguity. The result is shown in Figure 3.12.

The interesting result in Figure 3.12 is that the number of ambiguities for ‘n+1’ is the same as number of ambiguities for ‘n’, in the original grammar

Count	CPU	prodsaved	Trees
1	0	4	1
2	10	9	1
3	0	17	2
4	20	29	5
5	30	[illegible]	[illegible]
46	16290	17389	2257117854077248073253720
47	16350	18519	8740328711533173390046320
48	18060	19697	33868773757191046886429490
49	21160	20924	131327898242169365477991900
50	23040	22201	509552245179617138054608572

Figure 3.12 Test Result with ‘S’ Ambiguity Removed

(Table 3.6on page 61). There is no such relation between the number of productions saved for two consecutive values of n.

3.5.1.3 Eliminate All Ambiguities. Putting the above two steps together, the grammar is simplified to:

(1) S $\rightarrow$ NP VP
(2) NP $\rightarrow$ n
(3) NP $\rightarrow$ det n
(4) NP $\rightarrow$ NP PP
(5) PP $\rightarrow$ prep det n
(6) VP $\rightarrow$ v NP

There is no ambiguity in this grammar; the result is shown in Figure 3.13.

Count	CPU	prod	Trees
1	0	4	1
2	0	8	1
3	10	12	1
4	10	16	1
5	10	20	1
6	10	24	1
	70		
46	80	184	1
47	80	188	1
48	90	192	1
49	80	196	1
50	90	200	1

Figure 3.13 Test Result with All Ambiguities Eliminated

CHAPTER IV

PARSER GENERATOR

This chapter introduces a parser generator called Another Language Interpreter (ALI). I created ALI at first for disambiguating LL(1) grammars, namely removing left recursion and factoring the common left symbols. It evolved into a full parser generator. It was written with the target language ‘C’ in mind, but it was not too difficult to produce Lisp code with it. The current version can optionally produce code for both languages.

ALI is a YACC-like, Yet Another Compiler Compiler [YACC], approach for generating grammar tables for parsers, and specifying semantic actions.

ALI works with a very simple form of Backus-Naur notation (BNF) that does not use the alternative bar “|” for multiple production, “[...]” for optional symbols or "{...}” for repetitive symbols. For each production of a nonterminal, a separate rule must be present, but these rules do not have to be in sequence. The symbol "::=" separates the left hand side and the right hand side of productions. ALI lets the user insert comments in the input file. A comment starts with "!!" and continues to the end of line. A comment can appear anywhere in the input file that a blank symbol is valid.

Aside from the fact that ALI generates the LL(1) choice set for each production of the grammar as well as an efficient layout structure for grammar representation in both C and Lisp, it is capable of removing many types of LL(1) ambiguities for a given grammar. ALI is a stripped down version of a larger program that I wrote as part of my Ph.D Qualifying Exam. The original program was written in Pascal. It was intended to disambiguate grammars for LL(1) parsers and generate an LL(1) parse table with panic error recovery capability [Aho et al., 1986, p. 164]. To achieve the objectives, the following steps were taken:

1. Remove direct left recursion.
2. Factor the left common symbols of the same nonterminal productions. The factoring consists of:
   - 2.1. Direct factoring: In this case, one or more symbols are physically in more than one production.
   - 2.2. Indirect factoring: When there are common symbols in the choice set of two productions. This is more complicated than the above case.
3. Generating information data for error recovery, such as generator sets for each nonterminal. The generator set of a nonterminal is the set of all symbols that the nonterminal can generate. It is calculated by expanding the nonterminal productions, and then in turn in the right hand side of the production all nonterminals are expanded until we reach the end.
4. Reorganizing the productions of the same nonterminal to facilitate the error recovery, mostly for default production.
   A default production of a nonterminal is one of its productions that generates the minimum number of tokens amongst all of its productions.

The default production of a nonterminal is used to terminate the nonterminal with the least number of insertions by the error recovery process.

5. If the program is not to remove left recursion, productions of the left recursive nonterminals are reordered into two groups. The first group is the non-left recursive productions, and the second group is the left recursive productions. Note that there must be at least one non-left recursive production for any nonterminal, otherwise the nonterminal is considered to be *simply recursive* and *unparsable*. In this case, there is no parser, whatsoever, to parse the grammar including humans!
6. Generating the LL tables for the chart parser.

ALI is tuned for generating parse tables for natural language environments. Its objective is to generate parse tables that are easily accessed, in which the tables are tuned for the way the parser accesses them. For example, the internal code of the nonterminal is also the index to the nonterminal tables, as well as the symbol table. Since the internal codes are segmented into three parts: nonterminal, terminal, and action symbols, given the internal code of a symbol, its type is readily available.

4.1 Semantic Actions and Counter Actions

The interface between the parser and the semantic rules are action symbols. These are the user defined functions that will be executed as the parsing is in progress. ALI allows the placement of action symbols anywhere in the right hand side of the productions.

The semantic actions may be undone when there is a syntax or semantic error. Thus, each action may have a counter-action specified for it. In addition, the semantic action developer may insert arbitrary C, or Lisp, code into the file generated from ALI. ALI provides a declaration section to enable the developer to place declarations and definitions before the generated semantic actions.

4.2 Integer Coding

Since it is faster to work with integers than strings of characters, for each symbol of the grammar, ALI creates an integer code. The code starts with zero; zero is assigned to the first nonterminal symbol of the grammar. Code one is given to the next nonterminal. This process is repeated until all nonterminals have been assigned a code. Then codes are given to the terminal symbols. Last, action symbols are assigned a code. Note with this arrangement, when there are n nonterminals, m terminal symbols and p action symbols, then the following codes are assigned:

1. nonterminals are within range $< 0, n-1 >$,
2. terminals are in range $< n, n+m-1 >$, and
3. action symbols are in range $< n+m, n+m+p-1 >$.

Nonterminals are coded in a specific way, so within the nonterminal range there are sub-ranges:

1. Nonterminal with single productions. (if any).
2. Optional nonterminals (if any).

3. Recursive nonterminals (if any).
4. Nonterminal with multiple productions (if any).

Terminals are alphabetically sorted so the lexical analyzer can use a binary search instead of linear search for lookups.

This arrangement can be beneficial for error recovery, as well as for the use of this parser for syntax assisted editors.

4.3 Generated Tables

ALI produces four tables that contain all information about the grammar. The tables are designed to ease the access to various information about the grammar. The internal integer codes are used to index these tables. So as the grammar grows, only the size of these tables grows and search time remains constant, as we look in the m th cell of an array of size n.

4.3.1 Grammar Symbol Table. This is an array of grammar symbols. All references to the symbols of the grammar are indexed into this table, the internal code for the symbol. As mentioned, symbols are stored in this array in the following order:

- Nonterminals.
- Terminals.
- Action symbols.

The coding follows the convention outlined in the previous section.

4.3.2 Production Table. This is an array of production lists. Each list contains the right hand side of a production.

4.3.3 Nonterminal Table. Each nonterminal has an entry in this array. The index to this array is the internal code of the nonterminal. Each entry is a pair of indexes. The first one is an index to the first production of the nonterminal and the second one is an index to the last production of the nonterminal. Both indexes point to the production table.

4.3.4 LL(1) Choice Set Table. This table has a set of symbols for the LL(1) parser, one for each production. A nonterminal can be expanded into a production if and only if the input token is in the choice set table of the production. This table is parallel to the "Production Table", so the same index can be used for both of them.

For a nonterminal with multiple productions this set is essential, since it is the only way that the parser can select one production from many. Even for the single production nonterminals, it is useful to build this set since we can control the syntax errors as soon as they appear in the input file.

4.4 Grammar File Format

ALI accepts an input file *fileName.ext* and generates two files: *fileName.c* and *fileName.h*. For the Lisp version, *fileName.c* is used by another program to generate the tables in Lisp.

The grammar description input file must consists of five sections separated with “%%”:

1. declaration section.
2. nonkeyword terminal section.
3. keyword section.
4. grammar section.
5. function section.

Any of these sections could be empty, but if section four is empty, the file is meaningless.

4.4.1 Declaration Section. This section is used to port any definitions of variables, types, constants, etc. into the generated file before the actions. Actions are the inserted code in the grammar, surrounded by “{{ }}”, which will be executed as the parser gets to them. These definitions are used by the actions. The code is blindly copied into the generated file after the standard definitions generated by ALI. This section can be left blank.

4.4.2 Terminal Definition Section. This section is used to assign constant names to special keywords of the grammar. ALI considers any symbol that does not have a production to be a terminal symbol. Terminal symbols are divided into two sections:

1. Nonkeywords: These are the symbols that are recognized by the lexical analyzer as terminal symbols but they do not necessarily appear in the input file as the grammar symbols do. For instance, *identifier* is the type of many different symbols in a programming languages. In the natural language environment, many words are of part of speech *noun*. For instance, all of the following words are of type *noun*:

   stroke, reflex, return, vessel, baroreceptor
2. Keywords: These are the symbols that literally appear in the input file such as ‘(‘ or ‘:’.

4.4.2.1 Nonkeyword Terminal Definition. This section allows the user to generate internal code for nonkeywords of the language. The generated internal codes for these symbols are at the beginning of the terminal codes. Recall that in Section 4.2 we had the following range of codes:

1. nonterminals are in range < *0, n-1* >,
2. terminals are in range < *n, n+m-1* >, and
3. action symbols are in range < *n+m, n+m+p -1* >.

If there are *k* nonkeywords symbols the actual codes generated are as follows:

1. nonterminals are in range < *0, n-1* >,
2. nonkeywords are in range < *n, n+k-1* >
3. rest of the terminal symbols are in range < *n+k-1, n+m-1* >, and
4. action symbols are in range < *n+m, n+m+p -1* >.

The entries in this section are one line per symbol:

New symbol grammar symbol

For instance, if there is a terminal symbol ‘noun’, in the grammar, we could generate a constant symbol ‘N’ by entering a line:

N noun

which generates the C code:

#define N Internal-code-for-noun

This is useful for programming languages, where it is useful to separate the program keywords from the user defined identifiers. This section can be left blank.

4.4.2.2 Special Keywords Section. This section is used to assign constant names to special keywords of the grammar. This is a complementary to the previous section, for those keywords that are not acceptable as a constant identifier in 'C' like ':', this is an alternative approach for generating values/constant for symbols in the grammar. Note that ALI, unlike C, accepts any type of identifier, no restriction.

A constant name acceptable for 'C' is followed by the key word as in:

EQUAL =

COLON :

These names are used in the generated file with '#define' followed by the name and the internal code, an integer number, of the corresponding keyword. This section can be left blank.

4.4.3 Grammar Section. This section is the grammar itself with inserted semantic actions. Each action is a sequence of C or Lisp code surrounded by {{ }}. The code here is copied to the generated file as part of a switch statement. If another sequence of “{{some code}}” is followed by the first one, the second one is considered a counter action. A counter action is the reverse action taken by the parser to undo the action code when the parsing of a production fails after it has started.

Needless to say that as the user writes the action code, (s)he is responsible for the counter action code. On one hand, the parser executes the action code as the parse tree is being constructed. On the other hand, the parser executes the counter action codes as the parse tree is being destroyed.

There is no limit on the size of the code or the format of the code. The important requirement on the action code is that when the parser executes the corresponding action, it treats it as a boolean function. This requires the code to return a boolean result, so the parser can decide whether the action succeeded or failed.

4.4.4 Function Section. The function section contains any support functions needed by the actions. These are blindly copied into the generated output file. This section can be left blank.

4.5 Example

In this section, we present a file containing the grammar defined in Table 3.1.

The input file is named: gram.ali.

```
 /* Declaration section. */
#include <stdio.h> /* Standard I/O includes */
#include “gram.h” /* grammar include file */
%%
 !! No none-keyword definition.
%%
 !! No keyword definition.
%%
S ::= NP VP
S ::= S PP
NP ::= noun
NP ::= det noun
NP ::= NP PP
PP ::= prep NP
VP ::= verb NP
%%
/* Function definition section. */
```

CHAPTER V

SPELLING CORRECTION

5.1 Introduction

Spelling correction is an important task in an integrated parsing environment. A system cannot simply quit because of spelling errors, since they happen very often. The spelling correction process in an integrated parsing environment is not like the word processors that submit a few candidate words to the user to select one. It must provide an automatic correction mechanism and the response should be fast.

The significance of this research is that it brings the parser to aid spelling correction by using the context of the sentence. In most automatic spelling correction, such as the second version of Circsim-Tutor System, the spelling corrector returns one replacement word for each misspelled word. Interleaving the parser with the spelling correction process enables the spelling corrector to return a list of possible replacement words. The idea is nothing new. Humans use this technique to its fullest. Hall and Dowling [1980, p. 396] mention the use of the grammar in error correction.

The research extends the best match algorithm used in the second version of the Circsim-Tutor System, [Lee, 1990], to improve the word comparison. More important, it assigns error weights for the likelihood of the errors. The error weights allow the matching candidate words to be ranked. Lastly, it partitions the lexicon into 26 segments exploiting the fact that most errors do not appear at the initial character. However, it still can correct misspelled words that have a spelling error in the first character.

There are two basic steps in spelling correction: (1) detection of the error, and (2) automatic correction. I have limited the detection of spelling errors to isolated words. When the lexical analyzer cannot find a matching word in the lexicon for an input word, the word is considered misspelled. This is true for environments with a limited lexicon. When the lexicon is a full size dictionary, this is not a straightforward task. For instance, the Unix Spell program only detects errors.

In the following few sections we concentrate on word level spelling correction for finding replacement candidates for the misspelled word. After the replacement candidates are found, we need to decide what to do with them. Section 5.7 covers error correction using syntax structure.

5.2 Improved Spelling Correction Method

Realistic spelling correction cannot be done in isolation. First, the erroneous word is compared with the words in the lexicon. Second, the words that match with a minimum number of corrections are considered as candidates for replacement. Third, ranking of the candidates is done by sorting the error distance (how many letters of the misspelled words have to be changed to arrive at the candidate word). Finally, the syntax rules and the semantic rules are used to determine the best matching word. The following processes are involved:

1. Choice of words from the lexicon: This process decides what words of the lexicon are chosen and in what order they are compared with the erroneous word. For a small lexicon, a simple loop through all words from the lexicon suffice. For a large lexicon, some type of priority must be in place.
2. String comparison algorithm: This is the process that, given the erroneous word w_1 and the word from the lexicon w_2, makes the minimum number of changes to w_1 to transform it to w_2. This is the lowest level of the process that actually compares characters of the two words. It is the most important part of the spelling correction process since the rest of the system reacts to the result of this process. Succeeding sections of this chapter show how the introduction of the fourth way match to this process has improved its behavior and efficiency.
3. Ranking: As the words from the lexicon are compared with the misspelled word, the number of mismatched characters is saved. For each mismatched character, error weights are assigned. The matching words are ranked with respect to the error weight value.
4. Selection: The previous steps select the possible candidates for correction. This level uses the context of the sentence for possible corrections. The syntax, semantics and also phrase lookup (if one or more of the replacement candidates for the erroneous word happens to belong in a phrase combined with the next or previous word), can help narrow the number of candidates, we hope, to one. This supports the concept of minimal user intervention to correct a spelling error.

In general, the first and the second steps need some type of optimization when we are dealing with a lexicon with a large number of words. In the review of the literature, I discussed the concept of edit distance [Wu and Manber, 1992]. I add to the definition, *the keyboard effect of reversing characters*, as one correction. For instance, the correction of "*teh*" to "*the*", I count as one character reversal error although two corrections must be made to "*teh*" in most systems, either inserting *'h'* before '*e*' and deleting '*h*' at the end, or deleting '*e*' first and adding an '*e*' at the end.

I also extend the edit distance by assigning weights to each character correction. Instead of counting any type of error correction as one error distance, a different weight is assigned for different types of errors. This way, I distinguish between two candidates for the misspelled word even though they have the same edit distance.

Generally, an input word that cannot be matched with any word from the lexicon could be in any of the following categories:

1. General spelling errors: either the user does not know the correct spelling or it is a typographical error.
2. Abbreviations: after typing a few characters from a word, usually a long one, users often decide to quit. Sometimes they signal an abbreviation by putting a '.' at the end.
3. Word boundary: very often the space between two words is eliminated or an additional space is put in the middle of a word. Handling word

segmentation problems is important because it happens frequently. We will see that the employment of the error weight concept facilitates detection of word boundary problems.

5.3 Algorithm for Comparing Two Words

The spelling correction process is based on the comparison of the erroneous word with a word from the lexicon character by character. There are a large number of words to be compared with the erroneous word in a large lexicon. To reduce the time spent on comparing the erroneous word with irrelevant words from the lexicon, I put a limit on the number of mismatches, similar to that of Baeza-Yates and Gonnet [1992], with one exception. Instead of using an absolute limit on the number of allowable mismatches, I use a percentage of the length of the misspelled word. For example, "specification" and "sp **ice** fication" differ by two characters out of thirteen, so there is a 16% (2/13 x 100) disagreement. The words 'the' and 'tae' differ by one character out of 3, so they have a 33% or (1/3 X 100) disagreement.

The aim here is to terminate comparisons of the misspelled word with the irrelevant words from the lexicon as soon as possible. For instance, comparison of 'atiack' with 'choose' is terminated after mismatch of 'ati' and 'cho', and the process concludes that 'choose' is not a good candidate for replacing 'atiack'. As the two words are compared character by character, the error weights are calculated. As soon as the accumulated error weight reaches the disagreement limit the process is terminated.

Comparing the two words must be a fast process, as there are many words to be compared with a misspelled word. The comparison algorithm should be able to provide detailed correction steps to be applied to the misspelled word to transform it to the word that it has been compared with. This detail is used for ranking the candidates.

As a higher level optimization, some type of algorithm or at least heuristic should be used to lower the number of words selected for comparison. This will be discussed in detail in a later section.

5.3.1 Adaptive Disagreement Threshold. The disagreement threshold is used to terminate the comparison of an irrelevant word with the erroneous word. If the number is too high, we get a lot of irrelevant words. If it is too low, a lot of good candidates are discarded. A higher tolerance for error can affect the system efficiency as the comparisons are not terminated in early stages. For this reason, I came up with an adaptive method that dynamically lowers the tolerance for errors as better replacement candidates are found.

In order to have a wide acceptance capacity, I start with a 50% limit. As the first candidate is found with the value of disagreement n, the limit is dynamically set to the maximum of 30% and n. Of course, both upper bound and lower bound disagreement (50% and 30% mentioned here), are configurable. The whole idea is to have a system that, in the worst case, can find a replacement candidate, and as the candidates are found, it can dynamically look only for better candidates. After the first ten candidates are found (the number is configurable), the limit is set to the maximum value of disagreement amongst the candidates found so far. At any stage, if the disagreement is less than 10%, the process terminates and the comparison word will be considered the only replacement for the misspelled word.

5.3.2 Assignment of Error Weights. As the misspelled word and a word from the lexicon are compared character by character, for each type of character error, an error weight is assigned. The error weight is used to help selection of the best matching word between the two candidates. The lightest weight is for the character case. Also, in weight assignment, I distinguish between addition/loss of a character at the end of a word as compared to other positions in the word. This greatly helps in the recognition of abbreviated words and word boundary problems. The error weight for additional or missing characters at the end is raised back to regular additional/missing error weight, if the attempt to match the word for an abbreviation or split/join fails. Character substitution in case of keyboard effects,e. g., adjacent characters, as well as similar sounding characters, are also considered in the assignment of weights.

The following error categories are considered:

1. Character case : Most likely this is a keyboard effect or the user is trying to emphasize a point, as well as grammatical rules like capitalization of a word in the initial part of a sentence. This must not be considered an error at all, or at most, a very light error.
2. Reversed order : Most likely a keyboard effect.
3. Missing character(s) : Competence errors . Most likely the user does not know the right spelling (other than the keyboard effect when the user pushes a key softly and the character does not get inserted).
4. Missing character(s) at the end: there are four possibilities.
   - 4.1. Abbreviation: The user has typed ‘spec’ for ‘specification’.
   - 4.2. Word boundary problem: an extra white space is added in a word, resulting in a split. For example, “specifi cation” is typed for: “*specification* ”.
   - 4.3. Silent character: if the missing character is a silent one in the word from the lexicon, the error weight is reduced. Note, implementing this option requires the lexicon to provide such information for the lexical entries.
   - 4.4. A simple case of missing characters.
5. Added character(s): Competence errors . Most likely the user does not know the right spelling (other than possible keyboard effects when the user pushes a key by mistake, or the adjacent key to a character is also pushed, or a key is pushed for too long so that the character is repeated).
6. Added character(s) at the end: Extra character(s) at the end can signal a word boundary problem. The user types ‘thebook” in place of “the book”. Otherwise this could be a simple case of added character(s) as described in item 5 above.
7. Character substitution(s): Most likely a keyboard effect, or competence error.
   - 7.1. If the substitute key is the neighboring key of the character, then I

put much less error weight on the substitution character.

7.2. If the substitute character has a sound similar to that of the substituted character, the error weight is reduced.

A summary of the above categories and the error weights is given in Table 5.1. To clarify the concept, two columns are added to the table. The 'Erroneous word' column gives a misspelled word, and the column 'Matched with' gives a replacement word for the misspelled word when the correction is of the type listed in the 'Error Type' column.

Table 5.1 General Spelling Error Categories

Error Type	Erroneous Word	Matched with	Weight
Character case	**H**eart	**h**eart	3
Reversed order	t**eh**	t**he**	60
Missing character(s)	eror	er**r**or	80
Missing silent character	seudo	**p**seudo	25
Missing character(s) at the end	futur	futur **e**	12
Added character(s)	p**o**arser	parser	80
Added character(s) at the end	parser **e**	parser	36
Character substitution(s)	eff**a**ctive	eff**e**ctive	70
Adjacent character substitution	eff**r**ctive	eff**e**ctive	45
Similar sounding character substitution	**k**ool	**c**ool	45

In Chapter II, we saw that depending on the source of entry, the error pattern changes. For example, in a keyboard entry system, there is a high rate of keyboard adjacency substitution, e.g., substitution of the character 'D' for 'S'. In contrast, errors introduced by an optical character recognizer are based on confusion, due to featural similarities between letters (substitution of the character 'D' for 'O').

When the system concludes that the character 'c' at location n of the misspelled word is to be replaced by the character 'd' at location m of the word from the lexicon, the system checks for the possibility of adjacent keyboard characters. If the key for the character 'd' is not adjacent to the key for 'c' in the keyboard, then, the system checks to see if 'd' has a similar sound as 'c'. Two tables aid the spelling corrector in this regard:

1. lex_adjacent_key_list: this table is a list of pairs in which the first item in each pair is a character and the second entry is the list of the characters that are adjacent to the first character in a computer keyboard.
2. similar_sounding_keys_list: this table is a list of characters that sound the same. The table does not have to have an entry for all items. For instance,

there is no character that sounds similar to character ‘b’.

My system is designed to work with the keyboard entry system. If the system is used in an optical character recognizer, the table ‘lex_adjacent_key_list’ must be modified so each pair has the character as the first item in the pair, and all characters that look similar to the character. Table ‘lex_adjacent_key_list’ does not have to have an entry for all items in this case. For instance, in an optical character recognizer, probably no character has features similar to the character ‘A’. Therefore there is no entry for ‘A’ in the table.

As the system checks the similar sounding characters, when the check for adjacent characters fails, the content of the ‘similar_sounding_keys_list’ can also be changed to adapt the system for a new source of entry in addition to the ‘lex_adjacent_key_list’ table.

5.3.3 Comparison Details. As the misspelled word is compared with a word from the lexicon, each pair of characters in the two strings are compared four ways as shown in diagrams (1) to (4) in Figure 5.1. The type (4) comparison represents my own extension to the algorithm used in Lee [1990].

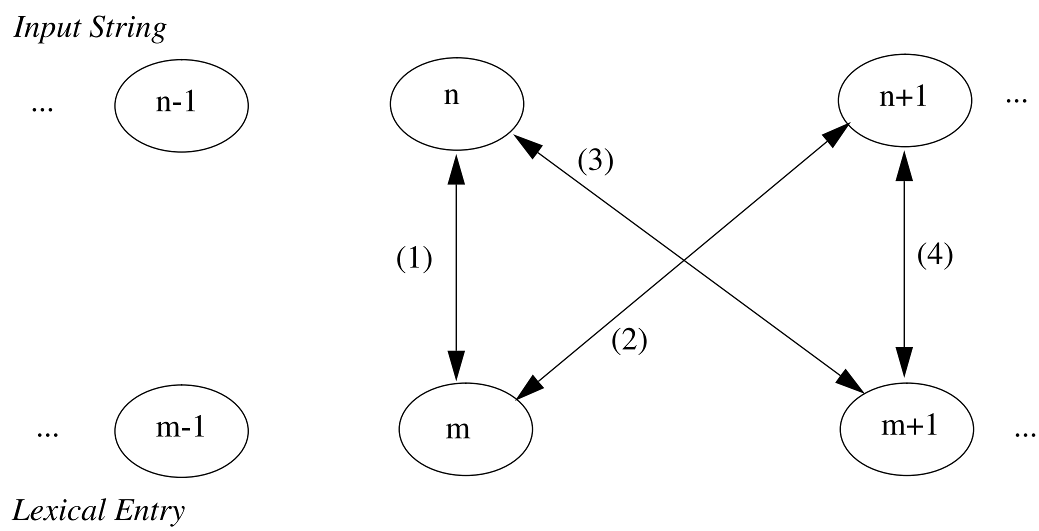


Figure 5.1 Character Comparison of Two Words

Table 5.2 shows the different comparison types. Character case errors, with a very small error weight, are omitted from Table 5.2. This situation is considered a special case of type A, no error.

In Table 5.2, ‘T’ Stands for True, ‘F’ for False, and ‘ —’ for don’t care. The order of comparison is important here. The algorithm first checks for condition (1). If (1) is true, we do not have any error, there is no need to do any spelling correction. If the only error is a case mismatch, the error weight of character case is assigned and the character case error is reported. If case (1) is not true, this signals a spelling error. The algorithm continues as follows:

If (2) and (3) are both True Then

Error Type “B” Exit

Table 5.2 Types of Spelling Comparison

type	Name of Comparison	Comparison Type 1	2	3	4
A	No error	T	—	—	—
B	Reversed order	F	T	T	—
C	**Character substitution for four way comparison**	F	—	—	T
D	Missing character	F	F	T	—
E	Added character	F	T	F	—
F	Character substitution	F	F	F	—

Else If (2) is False and (3) is True Then

If (4) is TRUE, we have the preliminary condition of 4th way, but the character at location 'n+1' of the input is different from the character at location 'm+2' of the word from the lexicon word, we have the complementary condition of 4th way match (this is fully described in Section 5.4 on page 84) then

Error Type "C", Exit

Else Error Type "D", Exit

End If

Else If (2) is True and (3) is False Then

If (4) is TRUE, we have the preliminary condition of 4th way, but the character at location 'n+2' of the input is different from the character at location 'm+1' of the word from the lexicon word, we have the complementary condition of 4th way match (Section 5.4 on page 84) then

Error Type "C", Exit

Else Error Type "E", Exit

End If

Else If (2) and (3) both are False Then

Error Type "F", Exit

End If

The error type (4) occurs when the n th character of the erroneous word does not match the m th character of the word from the lexicon, but the next two characters at locations n+1 and m+2 match. Later, in Section 5.4 we will see the reason for a four way match comparison. After the first check, (2) and (3) both being true, we simply check condition (4). Once (4) is True, regardless of the value of (2) or (3), we mark the error as type 'C'. Error Types 'C' and 'F' are basically the same.

As the two words are compared character by character, the increment of index *n* for the erroneous word and index *m* for the lexical entry is shown in Table 5.3. For example, if the error is an order reversal, we know that the comparison is done for two characters of both strings. Therefore, the next comparison will begin at the (n+2) nd character of the input string and the (m+2) nd character of the word from the lexicon.

Table 5.3 Index Movement in the Two Words to be Compared

Error Type	N	M	Comment
No Error	n+1	m+1	Advance the indexes for both words. Both characters in two word match.
Reversed order	n+2	m+2	Skip the next character for both, as the erroneous word current and the next character are to be transposed.
Character substitution for both type C and F	n+1	m+1	Advance to next one for both. Current character of erroneous word is to be replaced with the current character of the word from the lexicon.
Missing character	n+1	m+2	Copy the char(m) of the word from the lexicon to location n of erroneous word. Next, we need to compare next char. of erroneous word with char. at location m+2 of the word from the lexicon.
Added character	n+2	m+1	Extra character at n. Character n+1 matches character m.

In order to be able to report the details of the comparison between the two words, an structure called: *stringCompareValues* is created that contains a field for total error weight, and one field for each type of error defined in Table 5.1. As we start the comparison, all fields of s *tringCompareValues* structure are set to zero. When an error of type x is found, we first increment the total error weight by the corresponding value of error weight of x, defined in Table 5.1. Also, the associated counter of x in the structure is increased by one. So, at the end of the comparison, the structure contains the total error weight, and the exact number of corrections to be made to the erroneous word.

As an example, we consider correction of the erroneous word 'astack' in both three way match and four way match. For the erroneous word 'astack', compare the result of 3 way checking, the original algorithm, and my extension, an added fourth way check. Note that the possibility of a word boundary problem is checked and dismissed since *stack* and *tack* are not in our lexicon (otherwise one might say that 'astack' may be a garble for 'a stack', or 'as tack'.) In our limited lexicon, the erroneous word 'astack' generates the spelling correction candidates shown in Table 5.4. Column *weight* shows the total error weight. Character case problems are displayed in the column *case*. The reverse order of characters are shown in column *reverse.* Missing characters, except in the end of the string, are shown in column *miss* . Added characters, except in the end of the string, are shown in column *added* . Substituted characters are shown in column *subs* . Missing characters at the end of the string are shown in column *fewer* . Extra characters at the end of the string are shown in column *extra* . Finally, the column *Match word* , shows the text of the matching word from the lexicon.

Note that the five matching words do not include the word *attack* in Table 5.4.

Table 5.4 Candidates for ‘astack’ with the 3 Way Matching Algorithm

weight	case	reverse	miss	added	subs	fewer	extra	Match word
140	0	0	0	0	2	0	0	arrack
144	0	0	0	0	0	0	4	as
145	1	0	0	0	1	0	2	Asia
150	0	0	0	1	1	0	0	stuck
150	0	0	0	1	1	0	0	stock

Using the three way match algorithm to compare ‘astack’ with ‘attack’ generates the following result:

1. Matching ‘a’ with ‘a’ for n=m=1.
2. Declaring ‘s’ as extra as ‘t’ at location m=3 of the erroneous word ‘astack’ matches ‘t’ at location n=2 of word “attack”. The error weight is 80.
3. Missing ‘t’ at location m=4 of the erroneous word ‘astack’, as ‘a’ at 4’a’ at the location matches at location n=4 of word “attack”. The error weight is 80, giving a total of 160 for the weight of the errors.
4. The rest match.

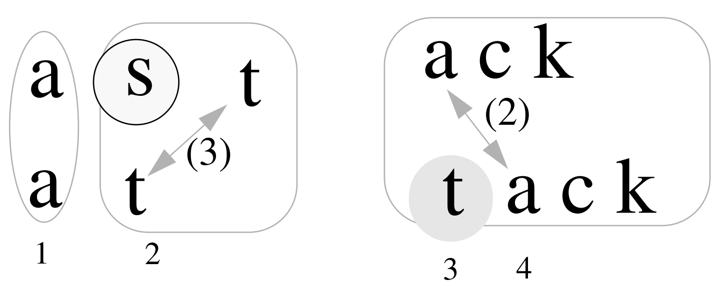


Figure 5.2 Three Way Comparison of ‘astack’ and ‘attack’

Using the four way match algorithm to compare ‘astack’ with ‘attack’, Figure 5.3, generates the following result:

1. Matching ‘a’ with ‘a’ for n=m=1.
2. Substituting ‘s’ with ‘t’ at location n=2 of the erroneous word ‘astack’ The weight of the error is 70.
3. The rest match.

Table 5.5 gives the result of the first top five matches using the four way match. The column headers are the same as the one described for Table 5.4.

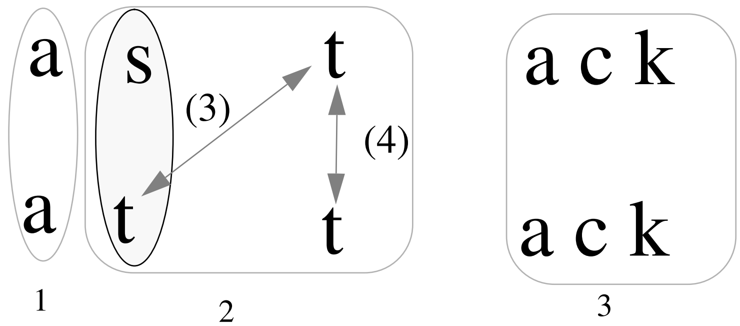


Figure 5.3 Four Way Comparison of ‘astack’ and ‘attack’

Table 5.5 Candidates for ‘astack’ with 3 Way Matching Algorithm

weight	case	reverse	miss	added	subs	fewer	extra	Match word
70	0	0	0	0	1	0	0	attack
106	0	0	0	0	1	1	0	attacks
140	0	0	0	0	2	0	0	attach
140	0	0	0	0	2	0	0	arrack
142	0	0	0	0	1	2	0	attacke d

5.4 Four Way Match

After we detect a mismatch between the character at location *n* of the misspelled word with the character at location *m* of the word from the lexicon, the next two characters at locations *n+1* and *m+1* of the two words are fetched for further analysis. Instead of comparing these four characters in three ways:

((n, m), (n, m+1), (m+1, n))

I also compare the character at location $n+1$ of the misspelled word with the character at location $m+1$ of the word from the lexicon. If they match, then my method differs from the original algorithm.

In this section we compare the three way match with the four way match. To compare the two methods, we introduce the term ‘best match.’ The best match is the process used when a human being makes a correction on an erroneous word. At every stage, we find the best match for an erroneous word and then compare its result with a three way match and a four way match to see which one does a better job.

With the four way match, we only differ from the original algorithm if (4) is

True. If (4) is False, we do exactly the same thing as in the original algorithm. The *reverse order* error is handled the same way in both the three way match, and the four way match method. Also, a *character substitution* error is handled the same way in both methods. The fourth match changes the three way matching algorithm for *missing character* and *added character* errors, since the fourth match is done before considering either one of these two errors.

In the following analysis, we need to be careful when referring to the characters past location ‘n+1’ of input, and ‘m+1’ of the word from the lexicon. Once the end of either of the two words is reached, then the end of a word does not match anything except the end of the word. In languages like C, this is easy. The end of a string is a null character, value zero in ASCII and it naturally does not match anything but another null character.

Section 5.4.1 analyzes the handling of *missing character* errors with and without four way match. Section 5.4.2 analyzes the handling of *added character* errors with and without four way match. Note that the assumption is that the condition for four way match:

- a mismatch between characters at location n of the misspelled word and location m of the word from the lexicon.
- the two characters at location n+1 and m+1 of the two words match.

is true, otherwise there is no difference between the two methods.

5.4.1 Handling of Missing Character Errors. When there is no match between the character at location ‘n’ of the input and the character at location ‘m’ of the word from the lexicon, and the input character at location ‘n’ matches the character ‘m+1’ of the word from the lexicon, a *missing character* error is found. This condition is shown in Figure 5.4.

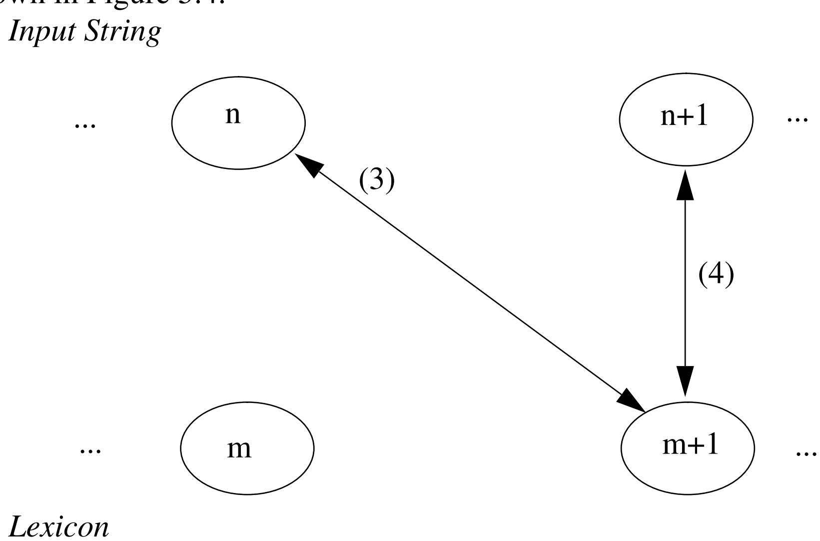


Figure 5.4 Character Relationships with a *Missing Character* Error

The original algorithm marks the character in the word from the lexicon at location ‘m’ as missing and advances the input character to the ‘n+1’ position and the word from the lexicon to ‘m+2’. Figure 5.5 illustrates the state of the comparison right after the first comparisons are done.

The solid lines indicate a test done in the past with a true result. Dotted lines indicate the test to be done in the next comparison. ‘N’ and ‘M’ are used here to have a similar comparison as in the original method in Figure 5.1.

The four way match method replaces the character at location ‘n’ of the misspelled word with the character at location ‘m’ of the word from the lexicon. As the characters at locations ‘n+1’ and ‘m+1’ of the two words match, it continues the comparison for characters at location ‘n+2’ and ‘m+2’.

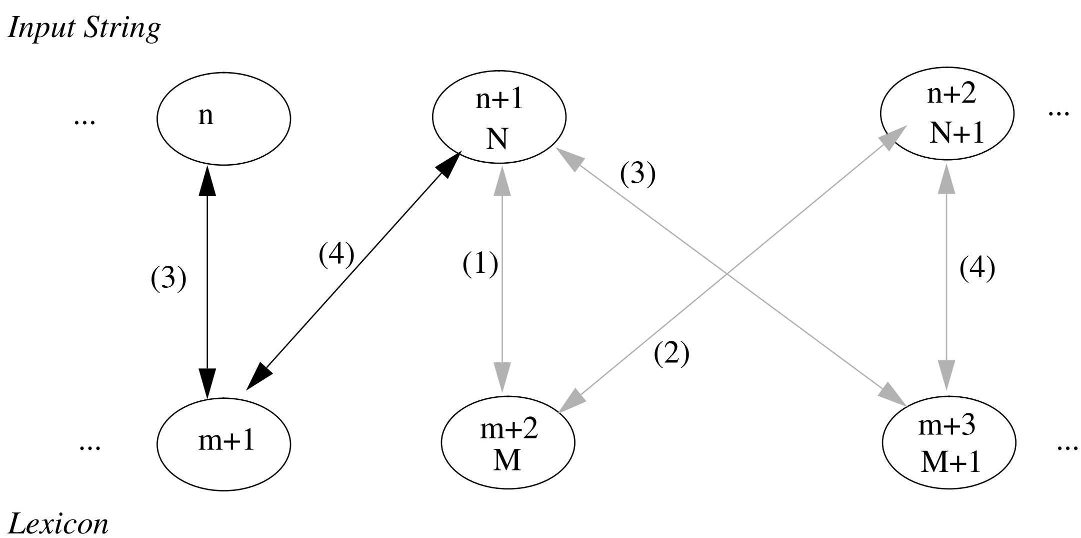


Figure 5.5 Comparison State after a *Missing Char.* in Three Way Match

Figure 5.6 illustrates the state of the comparison right after the first comparisons are done.

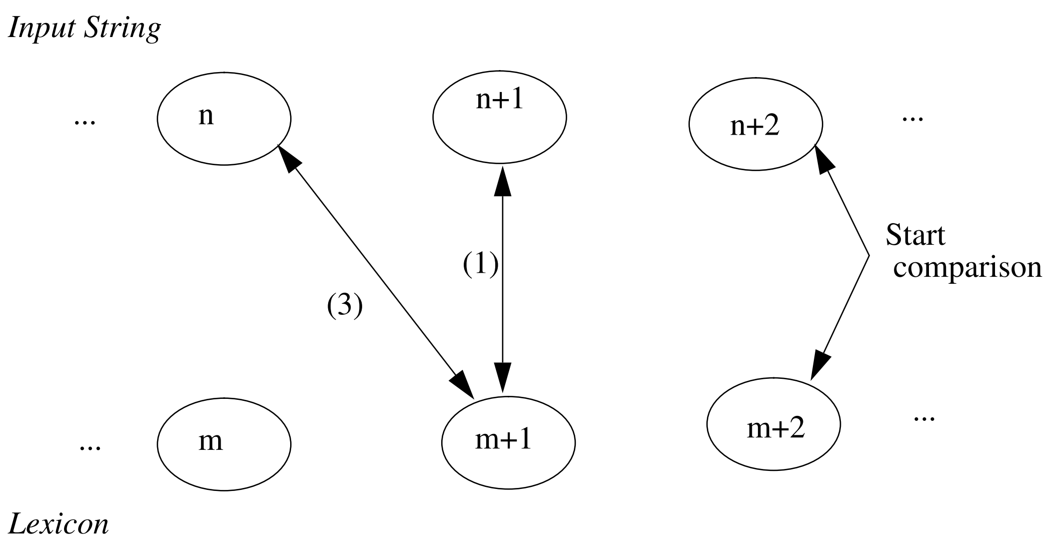


Figure 5.6 Comparison State after a *Missing Char.* in Four Way Match

To simplify the discussion, we refer to the character at location ‘n’ of the erroneous input word as character ‘n’ and to the character at location ‘m’ of the word from the lexicon as character ‘m’. We also use the indexes ‘n’, ‘n+1’, ‘n+2’ for the input word and the indexes ‘m’, ‘m+1’, and ‘m+2’, etc., for the word from the lexicon.

At this point, we analyze all possibilities of comparing characters ‘n+1’ and ‘n+2’ with the characters ‘m+2’ and ‘m+3’.

5.4.1.1 Type A: No Error. When the two characters at ‘n+1’ and ‘m+2’ match, we arrive at the situation displayed in Figure 5.7. In this case, the two characters at locations ‘n’ and ‘n+1’ of the input match the two characters at locations ‘m+1’ ‘m+2’. In fact, all four characters are the same, since originally we had a match between characters ‘n’ of input and ‘m’ of the lexicon, see Figure 5.7. This implies the two characters at location ‘m+1’ and ‘m+2’ are the same.

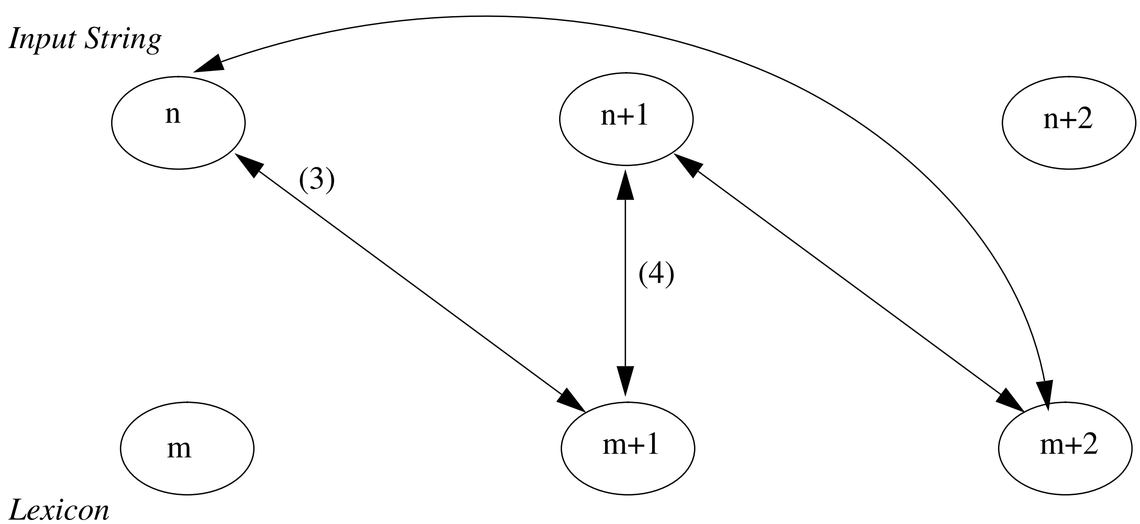


Figure 5.7 *Matching Character* after a *Missing character* Error.

The three way match does the following:

1. Determine that the character at location ‘m’ of the word from the lexicon is missing in the input word at location ‘n’.
2. Match the character at location ‘m+1’ of the word from the lexicon with the input word at location ‘n’.
3. Match the character at location ‘m+2’ with the character at location ‘n+1’.
4. Continue matching the characters at input location ‘n+2’ and the word from the lexicon at location ‘m+3’.

The four way match would do the following:

1. Substitute the character at location ‘n’ of the input with the character at location ‘m’ of the word from the lexicon.
2. Match the character at location ‘m+1’ of the word from the lexicon with input word at location ‘n+1’.
3. Continue matching characters at input location ‘n+2’ and the word from the lexicon at location ‘m+2’. Note that the characters at locations ‘n’ and ‘n+1’ of the misspelled words are the same as the character at location ‘m+1’ and ‘m+2’ of the word from the lexicon. Unless the character at location ‘n+2’ is

also the same as the previous two characters, the two characters at input location ‘n+2’ and the word from the lexicon at location ‘m+2’ do not match.

As an example, lets compare the words: ‘cooose’ and ‘choose’. The comparison of the second and third characters of ‘cooose’ with the third and fourth characters of ‘choose’ is shown in Figure 5.8. Let us try to compare the result of the three way match method with the four way match to correct the word ‘cooose’ to ‘choose’.

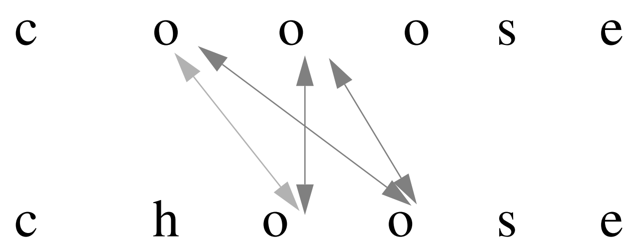


Figure 5.8 Comparison of ‘cooose’ and ‘choose’.

The three way match for comparing ‘coose’ and ‘choose’ (shown in Figure 5.9) does the following:

1. Match ‘c’ with ‘c’: n=m=1.
2. Determine that the character ‘h’ at location 2 of ‘c hoose’ is missing in ‘cooose’, therefore insert an ‘ **h** ’. We get ‘c **h** ooose’.
3. Match the character ‘o’ at location 4 of ‘c **h** ooose with the character ‘o’ at location 4 of ‘cho ose’.
4. Determine that the character ‘o’ at location 4 of ‘c **h** ooose is an extra character as ‘s’ in ‘c **h** ooose matches ‘s’ in ‘choo se’.
5. Rest match, (remaining ‘se’).

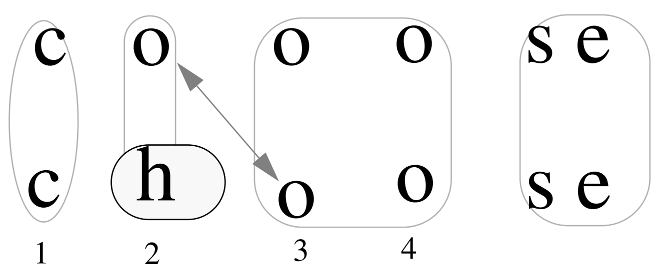


Figure 5.9 Three Way Comparison of ‘coose’ and ‘choose’

In summary we have inserted the character **h** at location 2, and remove d the character ‘o’ at location 4 of ‘cooose’.

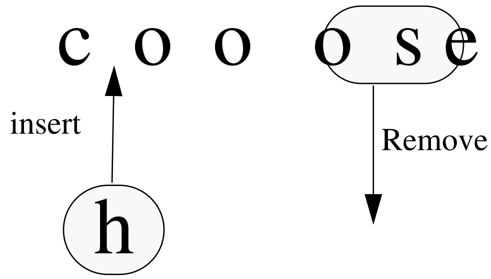


Figure 5.10 Correcting ‘cooose’ to ‘choose’ in the 3 Way Method

The four way match for comparing ‘cooose’ and ‘choose’ (shown in Figure 5.11) does the following:

1. Match ‘c’ with ‘c’: n=m=1.
2. Substitute the character ‘o’ at location 2 of ‘cooose’ with the character ‘h’ at location 2 of ‘choose’.
3. Rest match, (remaining ‘oose’).

Note that the four way match algorithm did a better job. It transformed ‘cooose’ to ‘choose’ with one correction, but the three way match corrected ‘cooose’ to ‘choose’ with two corrections. This is because all three characters ‘ooo’ in ‘cooose’ were the same.

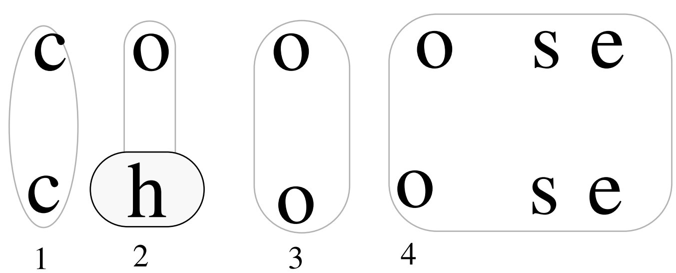


Figure 5.11 Four Way Comparison of ‘cooose’ and ‘choose’

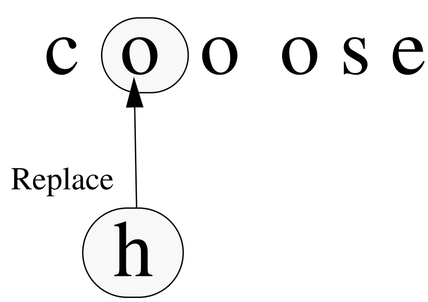


Figure 5.12 Correcting ‘cooose’ to ‘choose’ in the 3 Way Method

We now consider another example in which the three characters are not the same. Lets examine transforming ‘coose’ to ‘choose’ in both methods. The initial comparison diagram is shown in Figure 5.13.

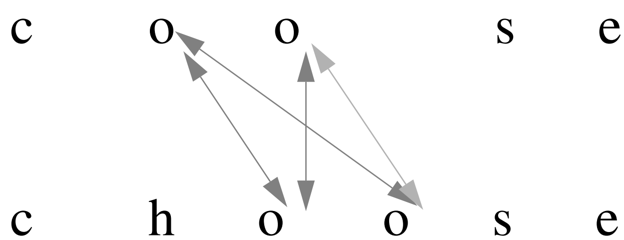


Figure 5.13 Comparison of ‘coose’ and ‘choose’

The three way match for comparing ‘coose’ and ‘choose’ (shown in Figure 5.14) does the following:

1. Match ‘c’ with ‘c’: n=m=1.
2. Determine that the character ‘h’ at location 2 of ‘choose’ is missing in ‘coose’, therefore insert an ‘**h**’. We get ‘c**h**oose’.
3. Rest match, (remaining ‘oose’).

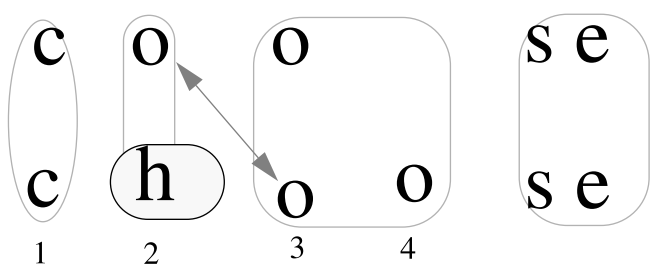


Figure 5.14 Three Way Comparison of ‘coose’ and ‘choose’

In summary we have inserted a character **h** at location 2.

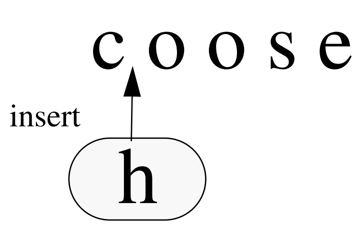


Figure 5.15 Correcting ‘coose’ to ‘choose’ in the 3 Way Method

The four way match for comparing ‘coose’ and ‘choose’ (shown in Figure 5.16) does the following:

1. Match ‘c’ with ‘c’: n=m=1.
2. Substitute the character ‘o’ at location 2 of ‘coose’ with the character ‘h’ at location 2 of ‘choose’: we get to ‘c**h**ose’.
3. Match the characters at location 3 of both words (character o).
4. Determine that the character ‘o’ at location 4 of ‘choose’ is missing in the

input word 'c **h**ose' at location 4, therefore inserts an 'o'. We get c **h**o**o**se.

5. Rest match. (remaining 'se').

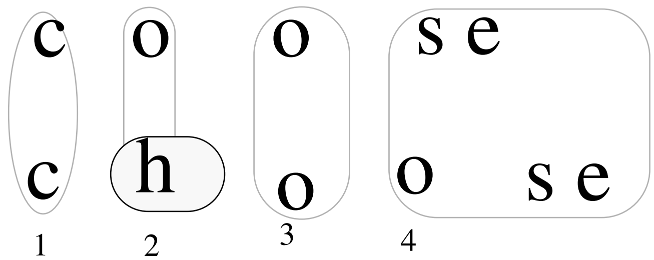


Figure 5.16 Four Way Comparison of 'coose' and 'choose'

So, using the four way match algorithm, we have replaced one character and inserted another to do the correction (shown in Figure 5.17).

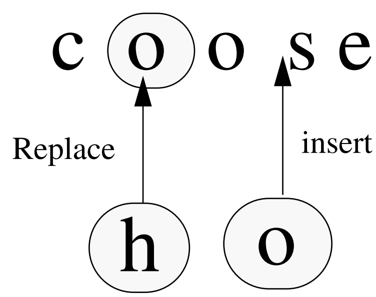


Figure 5.17 Correcting 'coose' to 'choose' in the 4 Way Method

In summary, one substitution and possibly another error is the result of incorporating the fourth match, which may be worse than the three way match. In other words, the characters 'n' and 'n+1' are equal and are the same as the characters 'm+1', and 'm+2'. Most likely, we have a missing character, the character at location 'm'. In this case, the four way match does not improve the situation, it creates a problem. So, this condition must be controlled in the four way match. In general, since the possibility of having three characters that are the same is not very high, we consider that the 3 way match does a better job.

5.4.1.2 Type B: *Reverse Order* Errors. Character 'n+1' matches the character 'm+3' and character 'n+2' matches the character at location 'm+2', as shown in Figure 5.18.

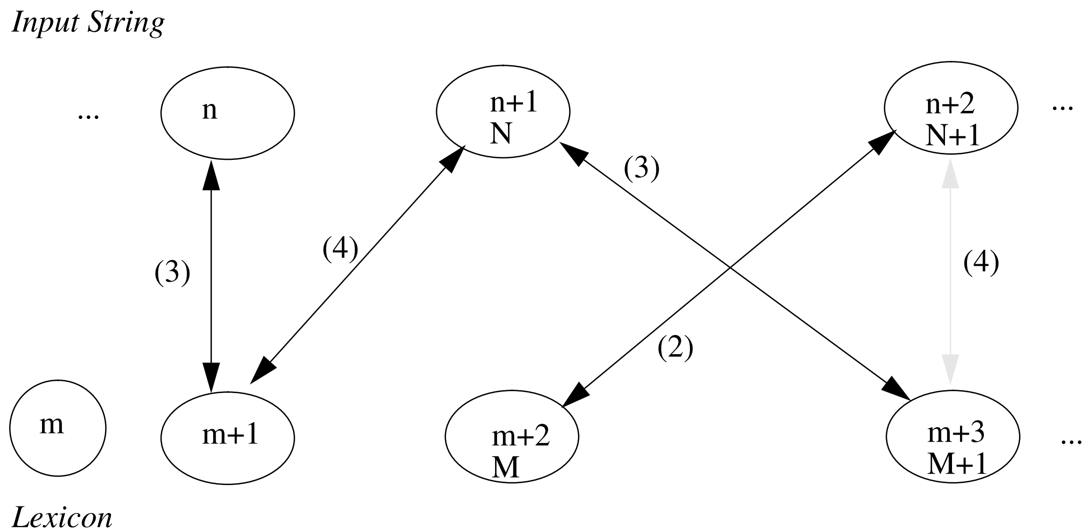


Figure 5.18 *Reverse Order* Error after a *Missing Character* Error

The three way match does the following:

1. Determine that the character at location ‘m’ of the word from the lexicon is missing as compared with the input word at location ‘n’.
2. Match the character at location ‘m+1’ of the word from the lexicon and the input word at location ‘n’.
3. Reverse the order of two input characters at locations ‘n+1’ and ‘n+2’.
4. Continue matching characters at input location ‘n+3’ and the word from the lexicon at location ‘m+4’.

The four way match does the following:

1. Substitute the character at location ‘n’ of the input for the character at location ‘m’ of the word from the lexicon.
2. Match the character at location ‘m+1’ of the word from the lexicon with the character of the input word at location ‘n+1’.
3. Match the character at location ‘m+2’ of the word from the lexicon with the character of the input word at location ‘n+2’.
4. Continue matching characters at input location ‘n+3’ and the word from the lexicon location ‘m+3’.

As an example, suppose that we have the erroneous input word ‘aalnce’ and the word ‘balance’ from the lexicon as shown in Figure 5.19.

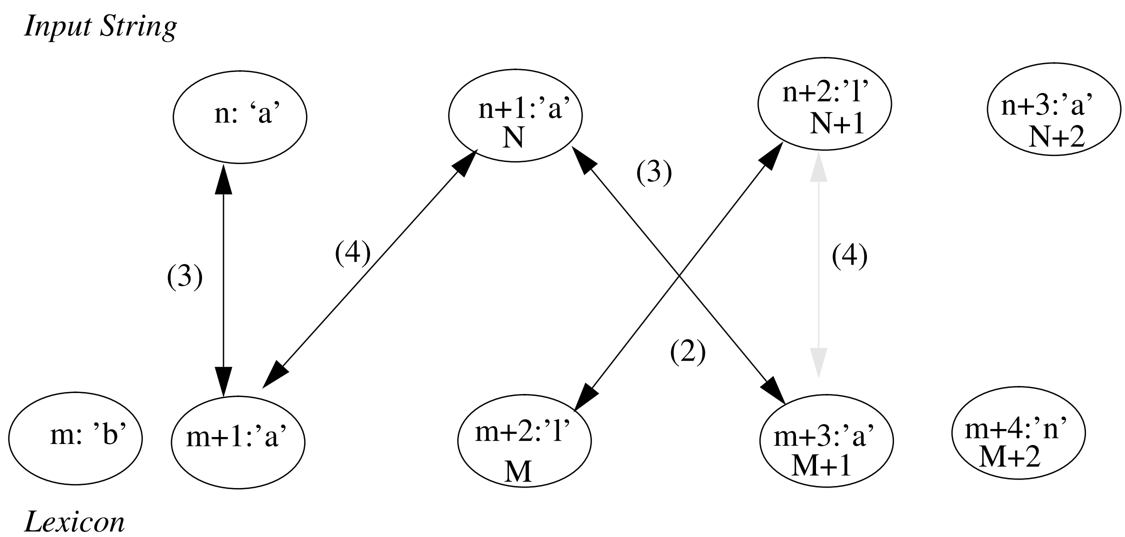


Figure 5.19 Matching ‘aalnce’ with ‘balance‘

The three way match does the following:

1. Determine that the character ‘b’ (from location m=1 of word: balance) is missing.
2. Match ‘a’ at location n=1 of the input and location m=2 of the word from the lexicon: balance
3. Reverse the order of the input characters at n=2 and n=3, ‘al’ to ‘la’.
4. The rest match.

The four way match does the following:

1. Substitute ‘a’ from location n=1 of input for ‘b’ at location m=1 of the word from the lexicon: balance.
2. Match ‘a’ at location n=2 of the input and at location m=2 of the word from the lexicon: balance.
3. Match ‘l’ at location n=3 of the input and at location m=3 of the word from the lexicon: balance.
4. Determine that the character ‘a’ (at location n=4) is missing, (to be copied from location m=4 of the word from the lexicon: balance).
5. The rest matches.

Another example: consider we have ‘aalance’ and the word from the lexicon ‘balance’, as shown in Figure 5.20.

The three way match does the following:

1. Determine that the character ‘b’ (from location m=1 of word: balance) is missing.
2. Match ‘a’ at location n=1 of the input and m=2 of the word from the lexicon: balance.

3. Reverse order ‘al’ of input at n=2 and ‘la’ of word from the lexicon at location 3.
4. Extra ‘a’ at location n=4 of the erroneous word.
5. The rest matches.

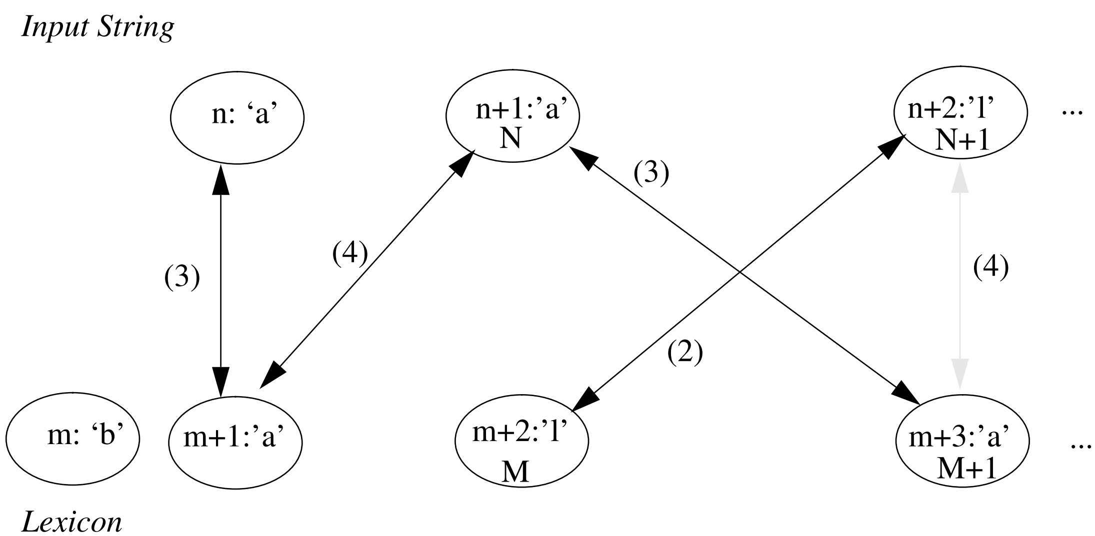


Figure 5.20 Matching ‘aalance’ with ‘balance‘

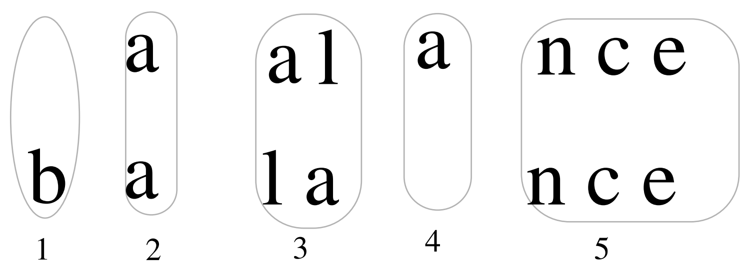


Figure 5.21 Correcting ‘aalance’ to ‘balance’ Using the 3 Way Method

The four way match does the following:

1. Substitute ‘a’ from location n=1 of the input with ‘b’ at location m=1 of the word from the lexicon: *balance* .
2. The rest matches.

5.4.1.3 Type D: ***Missing Character*** **Errors.** With a type D error, character ‘n+1’ of the input matches the character at location ‘m+3’, but, character ‘n+2’ of the input is different from the character at location ‘m+2’ of the word from the location. Three way match declares a missing character ‘m+2’, as shown in Figure 5.22.

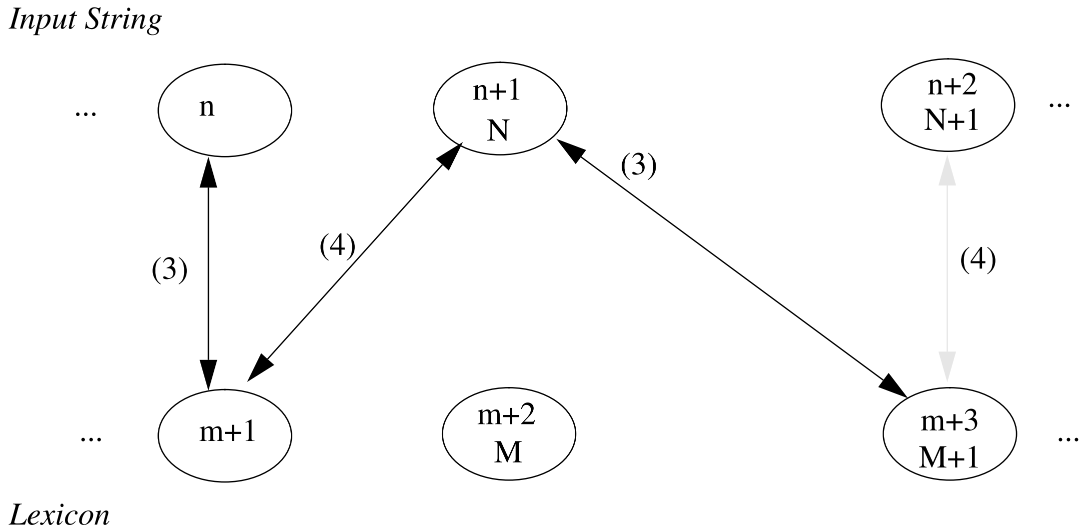


Figure 5.22 *Missing Char.* Error Followed by a *Missing Char.* Error

The three way match does the following:

1. Determine that the character at location m of word from the lexicon is missing, compared with the input word at location ‘n’.
2. Match the character at location ‘m+1’ of the word from the lexicon with the input word at location ‘n’.
3. Determine that the character at location ‘m+2’ of the word from the lexicon is missing as compared with the input word at location ‘n+1’.
4. Continue matching characters at input location ‘n+2’ and the word from the lexicon at location ‘m+4’, as shown in Figure 5.23.

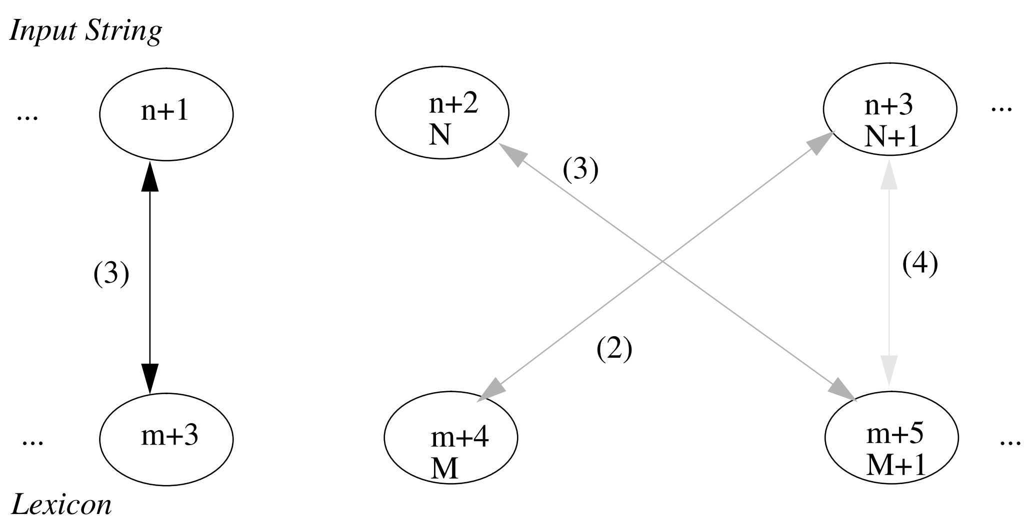


Figure 5.23 Next Comparison after a Type B Error after a Type D Error

The four way match does the following:

1. Substitute the character at location ‘n’ of the input with the character at location ‘m’ of the word from the lexicon.
2. Match up the character at location ‘m+1’ of the word from the word from the lexicon with the input word at location ‘n+1’.

At this point, the four way match algorithm compares the characters of the input at location ‘n+2’ and the word from the lexicon at location ‘m+2’. We know from the above that the characters at ‘n+2’ and ‘m+2’ do not match as we are in type D, and (2) is false. It all depends on the result of the comparison of characters at location ‘n+2’ and ‘n+3’ of the input and the characters at locations ‘m+2’ and ‘m+3’. It is known that character ‘n+2’ does not match ‘m+2’, so, we definitely have an error. All we know is that ‘m+3’ is the same as ‘m+1’ which is probably not very useful. So there is certainly an error here to be reported.

As an example, lets say that we have the erroneous word “cossnit” to be compared with the word “consist”, the relationship between the two word at location n=m=3 is as shown in Figure 5.24.

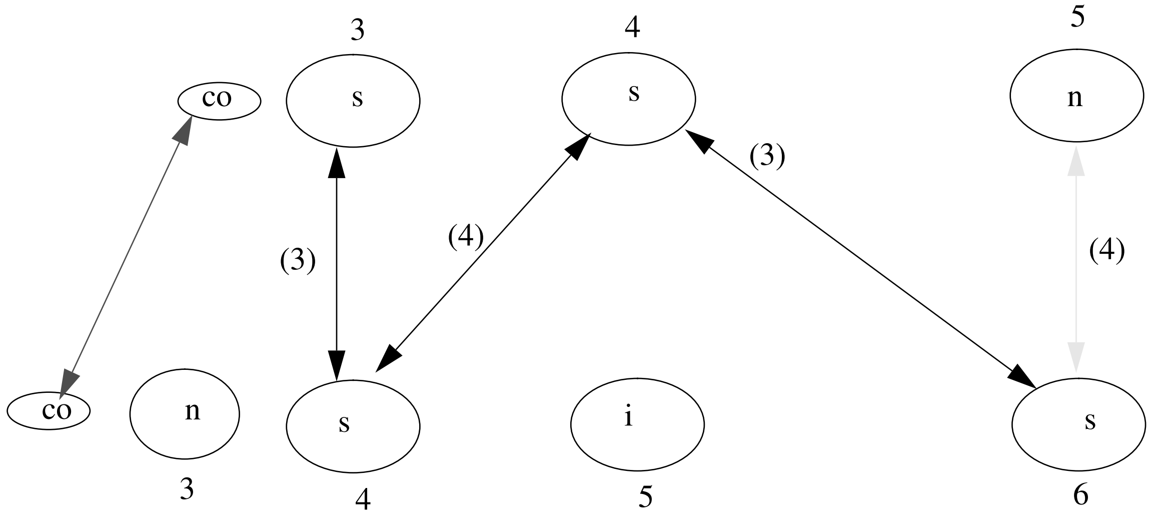


Figure 5.24 Relationship Between ‘cossnit’ and ‘consist at n=3

The three way match transformation is shown in Figure 5.25. Summary of the steps for three way match is:

1. match the first two characters of both words: “co”
2. insert an ‘n’ at location 3 of “cossnit”, transferring the word to “co **n** ssnit”.
3. insert an ‘i’ at location 5 of “co **n** ssnit”, transferring the word to “co **n** s **i** snit”.
4. replace ‘n’ at location 7 of “co **n** s **i** snit” to t, transferring the word to “co**n** s **i** s **t** it”.
5. drop the last two characters of “co **n** s **i** snit”, transferring the word to “co**n** s **i** s **t**”.

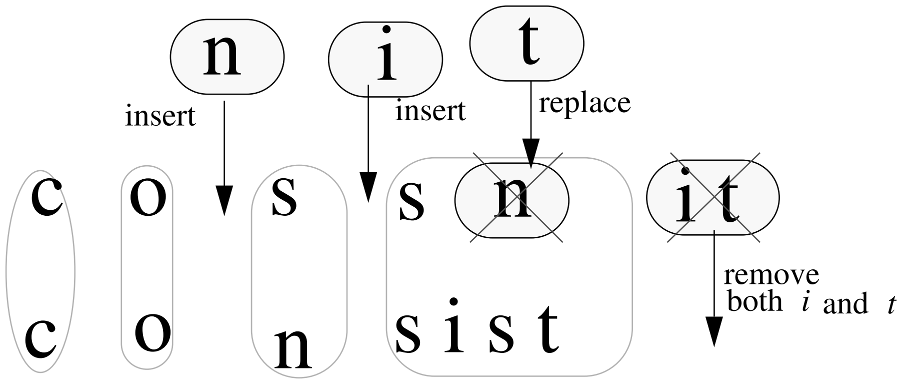


Figure 5.25 Three Way Match for ‘cossnit’ and ‘consist’

The four way match transformation is shown in Figure 5.26. Summary of the steps for four way match is:

1. match the first two characters of both words: “co”
2. replace ‘s’ at location 3 of “cossnit” wit n, transferring the word to “co **n** snit”.
3. Match the character ‘s’ at location 4 of both words.
4. drop ‘n’ at location 5 of “co **n** snit”, transferring the word to “co **n** sit”.
5. insert an ‘s’ at location 6 of “co **n** sit”, transferring the word to “co **n** si **s** t”.

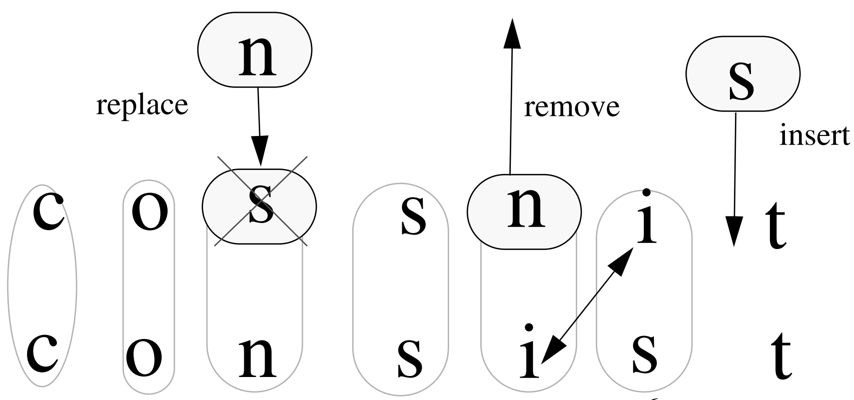


Figure 5.26 Four Way Match for ‘cossnit’ and ‘consist

In summary, the four way match corrects the word “cossnit” to “consist” with three modifications, whereas the three way match used five modifications to do the correction.

5.4.1.4 Type E: ***Added Character*** **Errors.** Given a type E error, the input character at location ‘n+2’ matches the character at location ‘m+2’ of the word from the lexicon. The three way match declares that there is an extra character ‘n+1’, Figure 5.27.

The three way match does the following:

1. Determine that the character at location ‘m’ of the word from the lexicon is missing as compared with the input word at location ‘n’.
2. Match the character at location ‘m+1’ of the word from the lexicon with the input word at location ‘n’.
3. Determine that the character at location ‘n+1’ of the input word is extra.
4. Continue matching characters at input location ‘n+3’ and location ‘m+3’.

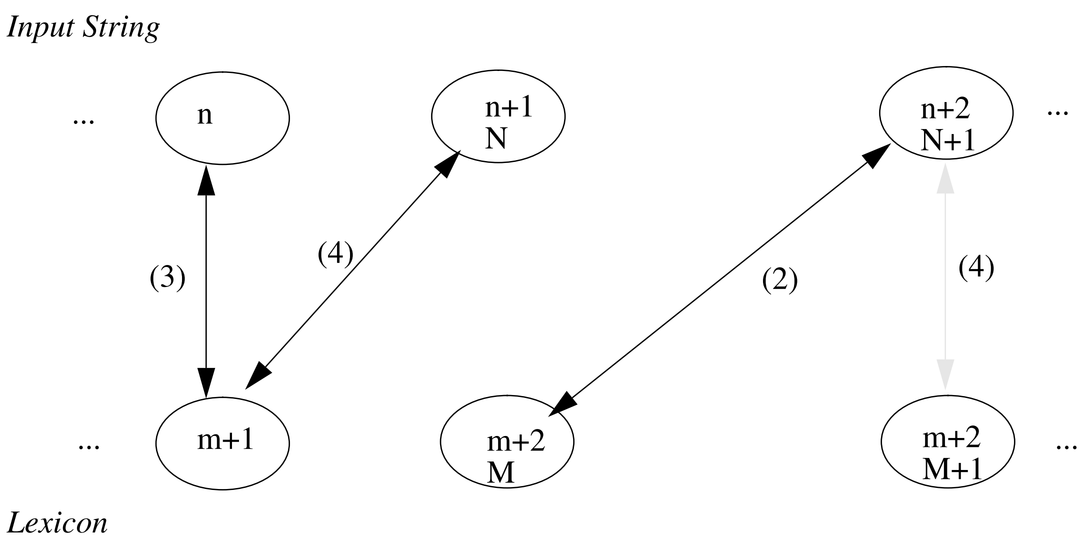


Figure 5.27 Type E Error after a Type D Error

The four way match does the following:

1. Substitute the character at location ‘n’ of the input with the character at location ‘m’ of the word from the lexicon.
2. Match the character at location ‘m+1’ of the word from the lexicon with the input word at location ‘n+1’.
3. Match the character at location ‘m+2’ of the word from the lexicon with the input word at location ‘n+2’.
4. Continue matching characters at input location ‘n+3’ with the word from the lexicon location ‘m+3’.

For example, the two words: “peessure”, and “pressure” fit this problem as shown in Figure 5.28.

Lets examine the steps for transforming the misspelled word: “peessure”, to “pressure”.

The three way match does the following:

1. Match the fist characters in both words.
2. Since the second character of “peesure” matches the third character of “pressure”, it determines that we have a missing ‘r’, therefore it inserts an **r**: “p**r**eessure”.

3. Characters at location three of both words: “p **r**eessure”, and “pr essure” match.
4. Character ‘ **e**’ at location 4 of “p **r**eesure”, does not match with ‘s’ at location 4 of “pre ssure”, but ‘s’ at location 5 of “p **r**eessure” matches ‘s’ at location 4 of “pre ssure”. Therefore, it drops ‘ **e**’ at location four “p **r**eessure”. We get to: “p**r**essure”.
5. Rest of the characters match.

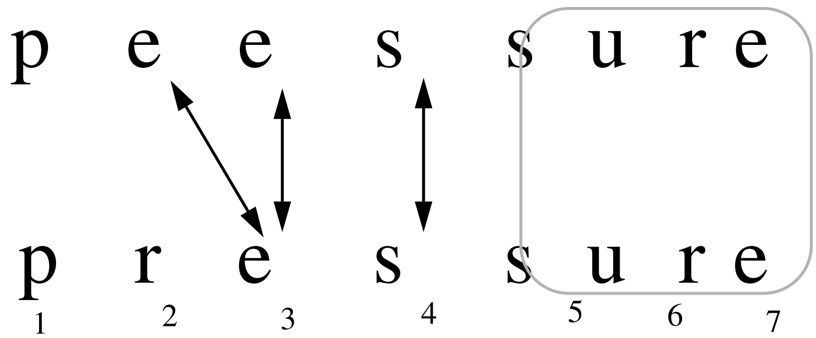


Figure 5.28 Comparison of ‘peessure’ and ‘pressure’

The four way match does the following transformation:

1. Match the fist characters in both words.
2. Character ‘ **e**’ at location 2 of “p eessure”, does not match with ‘r’ at location2 of “pressure”, but ‘e’ at location 3 of “pe essure” matches ‘e’ at location 3 of “pr essure”. Therefore, it replaces ‘e’ at location 2 to ‘r’: “p **r**essure”.
3. Rest of the characters match.

Note the four way match corrected the error with one modification to the word, but three way match did it with two modifications.

5.4.2 Handling of *Added Character* Errors. Given an *added character* error, Type E, (2) is True and (3) is False. In this case, the original algorithm marks the input character at location ‘n’ as an extra character and dumps it. The index of the input is advanced to ‘n+2’ and the index of the word from the lexicon is advanced to ‘m+1’, as shown in Figure 5.29.

The input at ‘n+1’ matches the character at location ‘m’ of the lexicon. For the next comparison, we have the diagram shown in Figure 5.19. Note again that the solid lines indicate a test done in the past with a true result. Dotted lines indicate the test to be done in the next comparison.

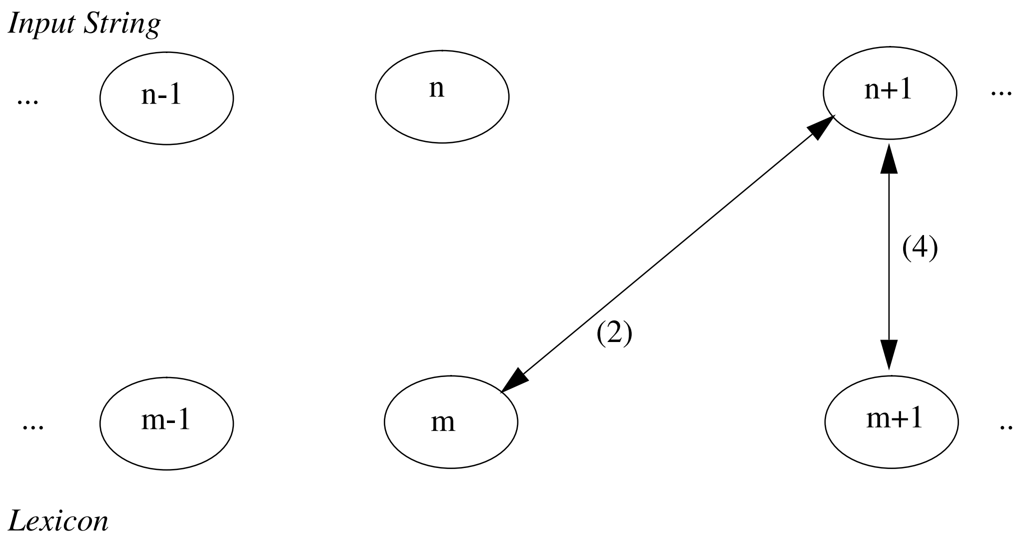


Figure 5.29 Status of Characters for a Type E Error with a Fourth Match

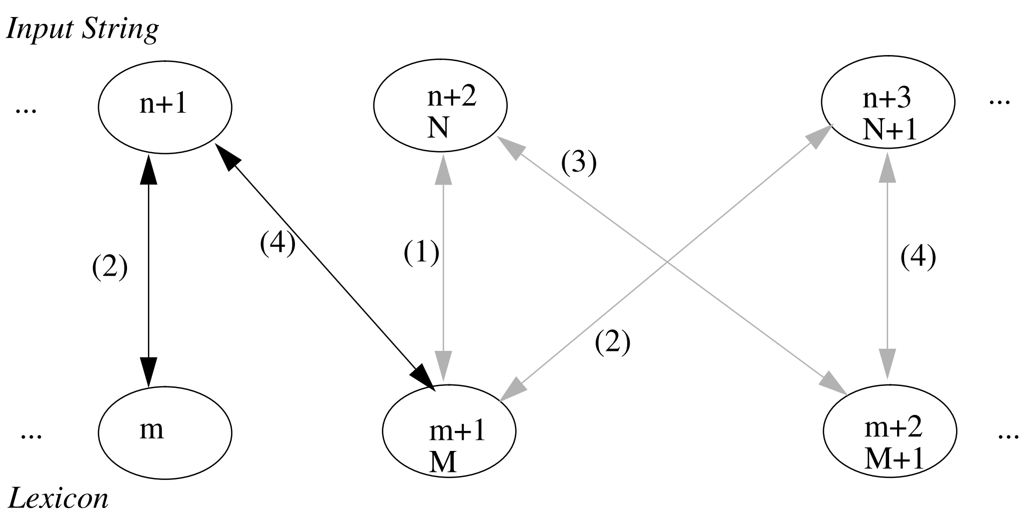


Figure 5.30 Comparison of Characters after a Type E Error

5.4.2.1 Type A: No Error. When the next two characters of the input word and the word from the lexicon match, (indicating no error: the input character at ‘n+2’ matches the character at ‘m+1’ of the word from the lexicon). This implies that the two characters at location ‘m’ and ‘m+1’ of the word from the lexicon are the same, as shown in Figure 5.31. This situation occurs in matching: ‘chioose’ with ‘choose’. The best match indicates an extra ‘a’.

The three way match does the following:

1. Determine that the character at location ‘n’ of the input word is extra.
2. Match the character at location ‘m’ with the input word at location ‘n+1’.
3. Match the character at location ‘m+1’ with the input word at location ‘n+2’.
4. Continue matching the characters at input location ‘n+3’ with ‘m+2’.

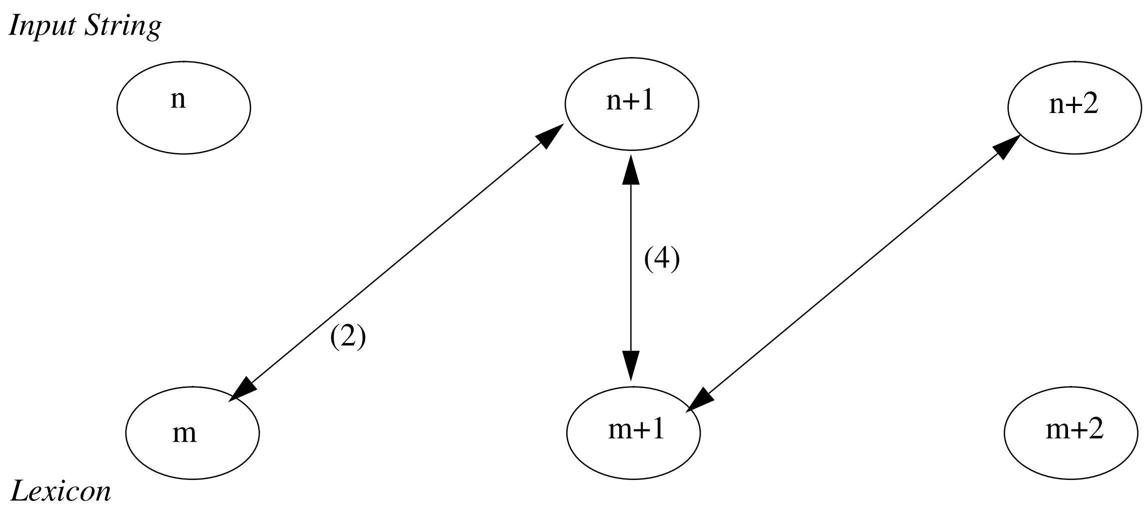


Figure 5.31 Matching Characters after an *Added Character* Error.

The four way match does the following:

1. Substitute the character at location ‘n’ of the input with the character at location ‘m’ of the word from the lexicon.
2. Match the character at location ‘m+1’ of the word from the lexicon and the input word at location ‘n+1’.
3. Continue matching the characters at input location ‘n+2’ and the location ‘m+2’. This test most likely will be false, since ‘n+2’ character matched ‘m+1’ of the word from the lexicon. In turn, the ‘m+1’ character of the lexicon was the same as the character at location ‘m’. It is highly unlikely that the third character would be the same. So there is an error here.

In the Type A condition, the characters of the input at locations ‘n+1’ and ‘n+2’ are equal and are the same as lexicon characters ‘m’ and ‘m+1’. Clearly we have an added character, the input character at location ‘n’. In this case the fourth match does not improve the situation, it creates problems. So this condition must be controlled in the four way match method.

An example would be the erroneous word “*chioose*” compared to “*choose*”, Figure 5.32.

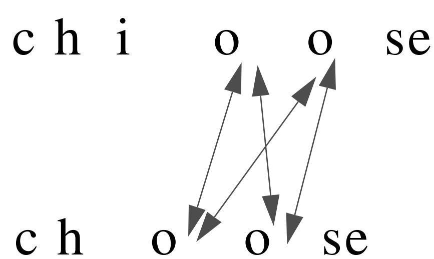


Figure 5.32 Comparison of ‘chiose’ and ‘choose’.

The three way match does the following (see Figure 5.33):

1. Match the first two characters ‘ch’ of the two words.

2. Mismatch of ‘i’ at n=3 with ‘o’ at m=3. But ‘o’ at location 4 of the erroneous word matches ‘o’ at location 3 of both words. This indicates an extra ‘i’ in the erroneous word.
3. Rest match.

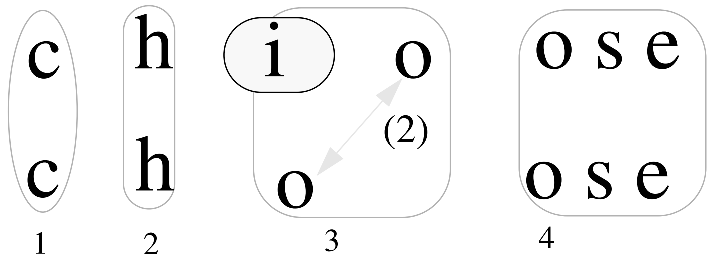


Figure 5.33 Three Way Comparison of ‘chioose’ and ‘choose’

The four way match (shown in Figure 5.34) does the following:

1. Match the first two characters ‘ch’ of the two words.
2. Mismatch of ‘i’ at n=3 with ‘o’ at m=3. But ‘o’ at location 4 of the erroneous word matches ‘o’ at location 3 of both words. Also ‘o’ matches ‘o’ at location 4 of both words. This would cause a call for substituting ‘i’ with ‘o’. The indexes are set to n=m=5 as the character ‘o’ at location 4 of both words matches.
3. Mismatch of ‘o’ at n=5 with ‘s’ at m=5. But matching ‘s’ at location 6 of the ‘chioose’ word matches with ‘s’ at location 5 of the ‘choose’ word. This indicates an extra character ‘o’ of the erroneous word.

Rest match.

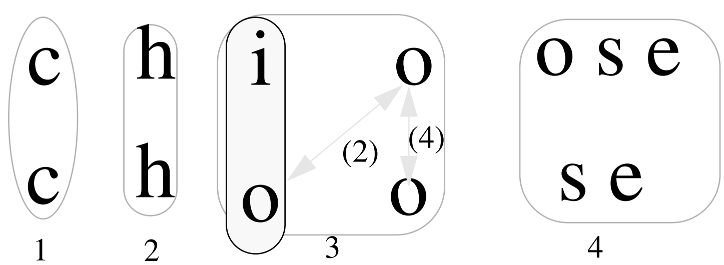


Figure 5.34 Four Way Comparison of ‘chioose’ and ‘choose’

In summary we have:

- replaced ‘i’ to ‘o’ at n=3.
- dropped ‘o’ at location n=5.

In conclusion, the three way match corrects ‘chioose’ to ‘choose’ with an extra character ‘i’ at location 3. But, the four way match corrects it with substitution of ‘i’ to ‘o’ at location 3, and also an additional ‘o’ at location 5. The weight of error in the three way match is 80. In the four way match the error weight is (90 + 80 = 170), so the system prefers the three way match result.

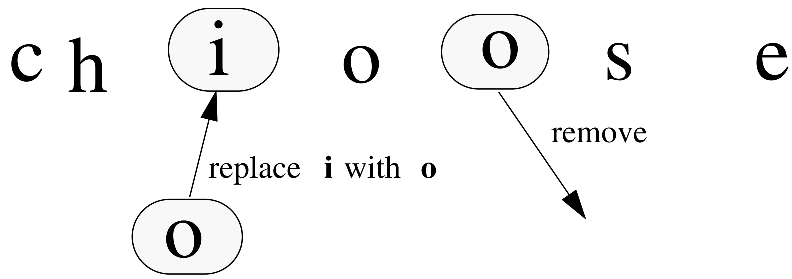


Figure 5.35 Correcting ‘chioose’ to ‘choose’ in the 4 Way Method

5.4.2.2 Type B: *Reverse Order* Errors. Given a Type B error, the input character ‘n+2’ matches the character ‘m+2’ of the word from the lexicon. Also, the input character at location ‘n+3’ matches the character at location ‘m+1’. This implies that in the word from the lexicon, at least two characters at location ‘m’ and ‘m+1’ are the same as they both are the same as the input character at location ‘n+1’ as shown in Figure 5.36.

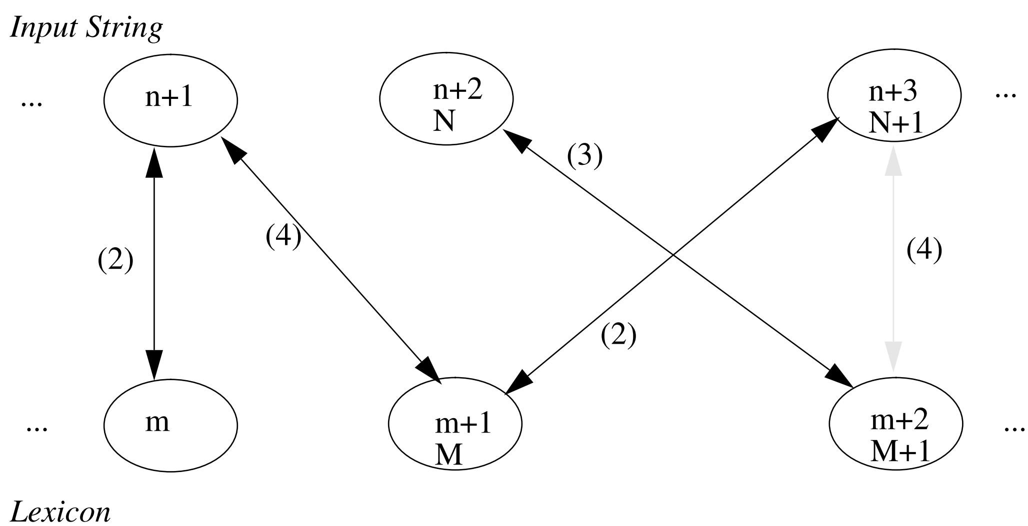


Figure 5.36 *Reverse Order* Error after an *Added Character* Error

We know the two characters at locations ‘m’ and ‘m+1’ are the same. The three way match does the following:

1. Find an extra character in input word at location ‘n’.
2. Match the character at location ‘m’ with the input word at location ‘n+1’.
3. Reverse the order of input characters at locations ‘n+2’ and ‘n+3’.
4. Continue matching characters at input location ‘n+4’ and the word from the lexicon at location ‘m+3’.

The four way match does the following:

1. Substitute the character at location ‘n’ of the input for the character at location ‘m’ of the lexicon word.
2. Match the character at location ‘m+1’ of the word from the lexicon and the input word at location ‘n+1’.
3. Match the character at location ‘m+2’ of the word from the lexicon and the input word at location ‘n+2’.
4. Continue matching the characters at input location ‘n+3’ and the word from the lexicon at location ‘m+3’.

The two words ‘frieze’ and ‘freeze’ fit in this type of problem as shown in Figure 5.37.

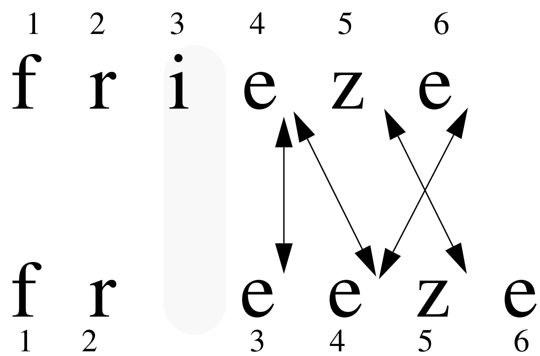


Figure 5.37 Comparison of ‘frieze’ and ‘freeze’

The three way match does the following:

1. Match the first two characters ‘fr’ of the two words.
2. There is a mismatch for third characters of the two words. Since the fourth character of “frieze”, matches the third character of “freeze”, it determines that we have an extra ‘**i**’ at the third position, therefore it drops the ‘**i**’: “freze”
3. Characters ‘ze’ at position 4, and 5 of “freze”, are in reverse order of characters ‘ez’ at position 4, and 5 of “freeze”, therefore reverse their order: “freez”
4. Determine that the last character of ‘freeze’ is missing in ‘freez’, therefore insert it: freeze.

The four way match, shown in Figure 5.38, does the following transformation:

1. Match the first two characters ‘fr’ of the two words.
2. There is a mismatch for third characters of the two words. Since the fourth character of “frieze”, matches the fourth character of “freeze”, it substitutes the third character of “frieze” with the third character of “freeze”. Therefore we have: “fr**e**eze”.
3. Rest match.

In summary, three way match transformed ***frieze*** to ***freeze*** with three corrections. The four way match did the correction with one replacement.

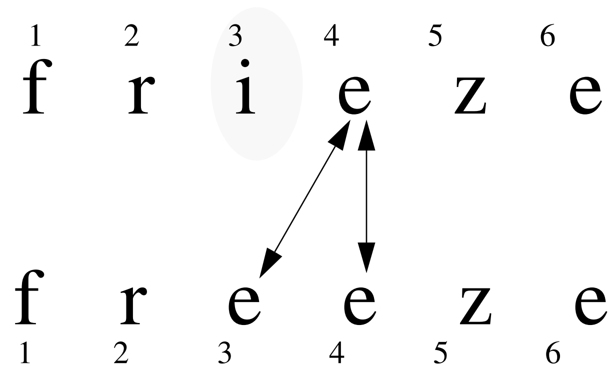


Figure 5.38 Four Way Comparison of 'frieze' and 'freeze'

Considering the fact that it takes three corrections to transform the misspelled word: ***frieze*** to the irrelevant word ***: tries***, it is obvious that the result of the three way match is misleading. Note that the ranking process takes the result of the comparison and based on that sorts the replacement words for the misspelled word. Obviously, 'freeze' is a better replacement for 'frieze' than 'tries'.

5.4.2.3 Type D: ***Missing Character*** **Errors.** With a type D error, *missing character*, the input character 'n+2' matches the character 'm+2' of the word from the lexicon, but the input character at location 'n+3' is different from the character at location 'm+1' of the word from the lexicon, as shown in Figure 5.39. We know that the two characters at locations 'm' and 'm+1' are the same.

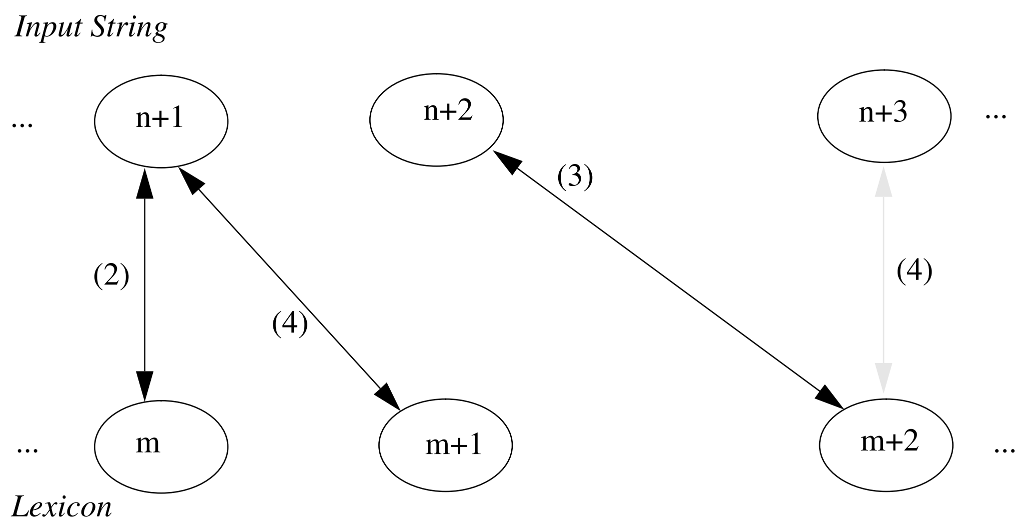


Figure 5.39 *Missing Char.* Error after an *Added Char.* Error

The three way match does the following:

1. Find an extra character in input word at location 'n'.
2. Match the characters at location 'm' of the word from the lexicon and the input word at location 'n+1'.

3. Identify the missing character ‘m+1’ of the word from the lexicon right before the input characters at locations ‘n+2’.
4. Match the characters at location ‘m+2’ of the word from the lexicon and the input word at location ‘n+2’.
5. Continue matching the characters at input location ‘n+3’ and location ‘m+3’ of the word from the lexicon.

The four way match does the following:

1. Substitute the character at location ‘n’ of the input with the character at location ‘m’ of the word from the lexicon.
2. Match the character at location ‘m+1’ of the word from the lexicon and the input word at location ‘n+1’.
3. Match the character at location ‘m+2’ of the word from the lexicon and the input word at location ‘n+2’.
4. Continue matching the characters at input location ‘n+3’ and location ‘m+3’ of the word from the lexicon.

For example, lets try to correct “camnon” to “cannon”. The three way match, shown in Figure 5.40, does the following:

1. Remove m at the third position: “canon”.
2. Reverse order characters at location 4 and 5: “can **no**”
3. Insert an ‘n’ at the end: “can **non**”.

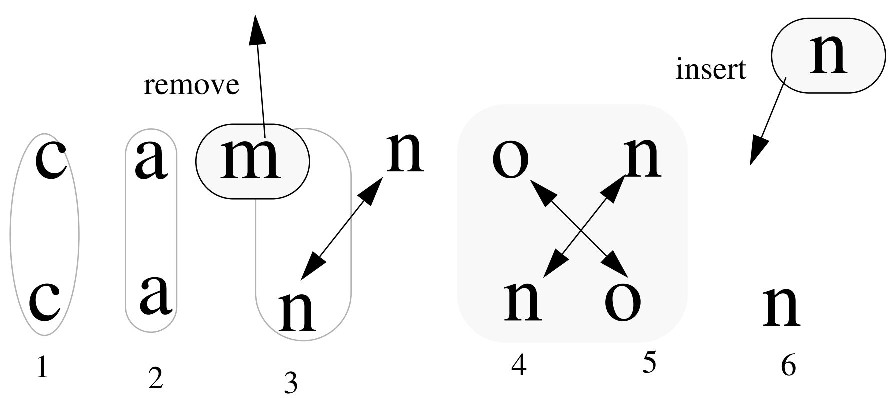


Figure 5.40 Three Way Matching of ‘camnon’ and ‘cannon’

The four way match (shown in Figure 5.41) does:

1. Replace the third position character of: “camnon” with the third letter in ‘cannon’. Note the fourth position characters in both words match: “ca**n**non”.
2. Rest match.

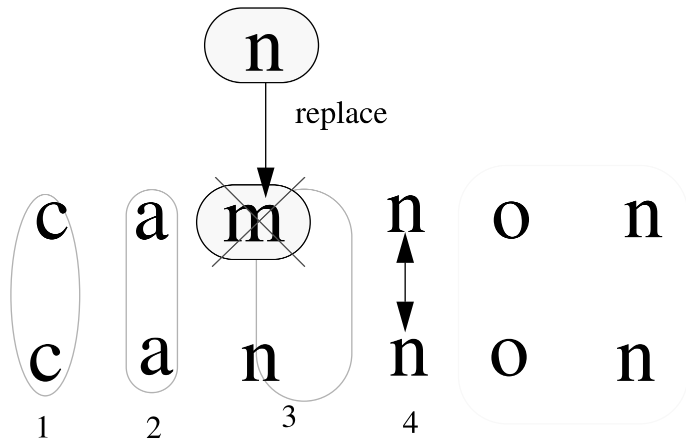


Figure 5.41 Four Way Matching of 'camnon' and 'cannon'

5.4.2.4 Type E: ***Added Character*** **Errors.** Given a type E error, *added character* , the input character at location 'n+3' matches the character at location 'm+1' of the word from the lexicon. The input character at location 'n+2' is different from the character at location 'm+2' of the word from the lexicon, as shown in Figure 5.42. We know that the two characters at locations 'm' and 'm+1' are the same.

The three way match does the following:

1. Find an extra character in the input word at location 'n'.
2. Match the character at location 'm' of the word from the lexicon and the input word at location 'n+1'.
3. Find an extra character at location 'n+2' of the input.
4. Match the character at location 'm+1' of the word from the lexicon and the input word at location 'n+2'.
5. Continue matching the characters of the input at location 'n+4' and location 'm+2' of the word from the lexicon.

The four way match does the following:

1. Substitute the character at location 'n' of the input with the character at location 'm' of the word from the lexicon.
2. Match the character at location 'm+1' of the word from the lexicon and the input word at location 'n+1'.
3. The characters at location 'm+2' of the word from the lexicon and the input word at location 'n+2' do not match.
4. It is known that 'n+2' does not match 'm+2'. So, there is certainly an error here to be reported. At this point, the four way match compares the pair of characters of the input words at locations 'n+2' and 'n+3', with the characters at locations 'm+2' and 'm+3' of the word from the lexicon. Figure 5.43 shows the comparison state of characters at locations 'n+2' and 'n+3' of the misspelled word with the characters at locations 'm+2' and 'm+3' of the word from the lexicon.

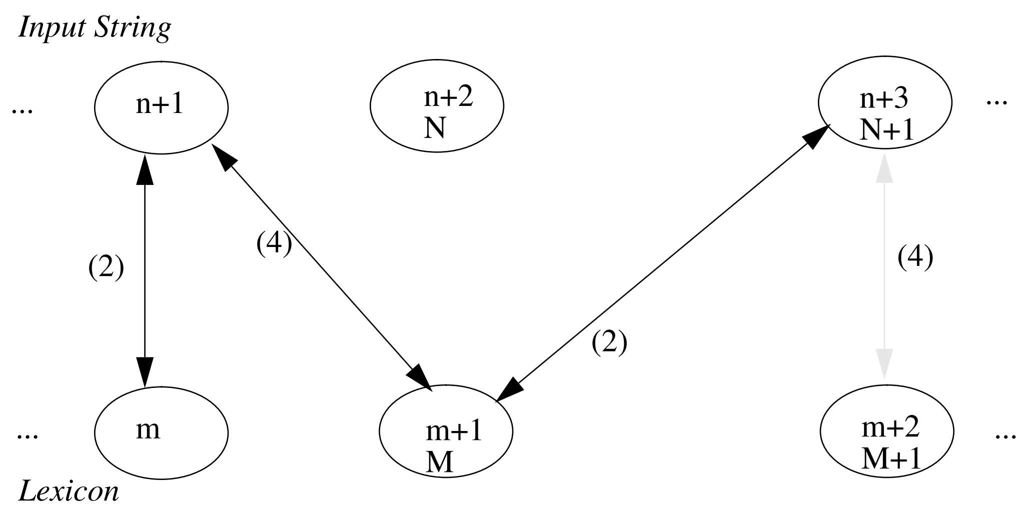


Figure 5.42 Added Char. Error after another Added Char. Error

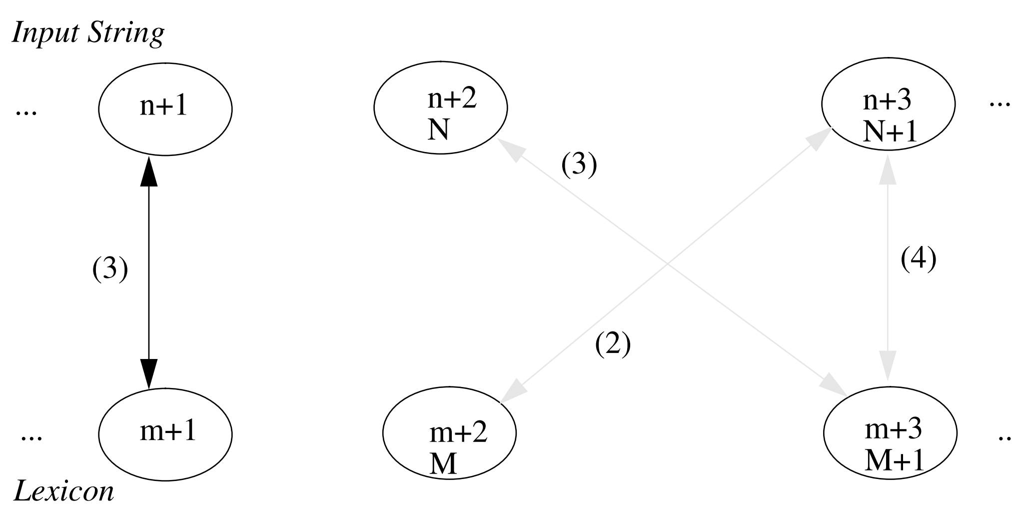


Figure 5.43 Four Way Comp. of an Added Char.

For example, lets try to correct "chioeo" to "choose". The comparison diagram is shown in Figure 5.44.

The three way match does the following:

1. drop third position character 'i': "choeo".
2. drop fourth position character 'e': "choo".
3. insert "se" at the end: "choose"

The four way match does the following:

1. substitute third position character 'i' to o: "ch **o**oeo"
2. substitute the last two characters "eo" to "se" (by inserting an s and dropping o): "ch **o**o**s**e".

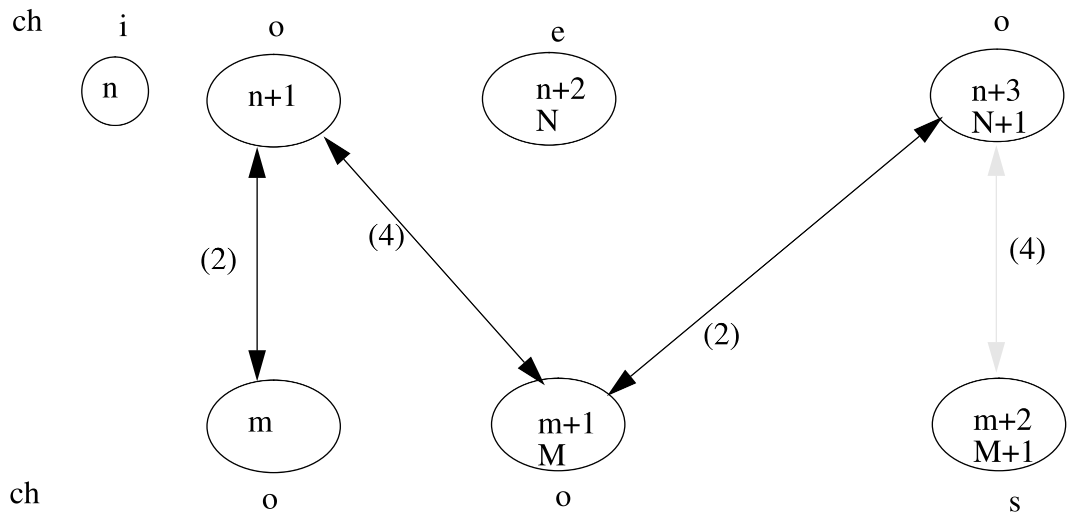


Figure 5.44 Comparison Diagram for “chioeo” and “choose”

5.4.3 Analysis. In this section, we put the results of the study together for further analysis. Using the analysis, we combine the three way match algorithm and the four way match algorithm to produce a better spelling correction algorithm.

Note that the error occurred when the character at location n of the erroneous word (referred to as char(n)), did not match the character at location m of the lexicon word (referred to as char(m)).

The study looked at the way that both methods keep working following the correction at char(n). Our assumption here is that char(n+1) = char(m+1). Otherwise, there is no four way match.

In summary, we ignore error types A, B and C (refer to Table 5.2 on page 81, for error types). Since the error type F is the same as the error type ‘C’, it is ignored too. So, for the consequence matching, we are only interested in a three way match for handling errors type D and E. First consider errors of type D: *Missing Characters*, shown in Table 5.6. Next, consider errors of type E: *Added Characters*, shown in Table 5.7.

Summarizing the result, described in the two tables, Table 5.6 and Table 5.7, leads us to Table 0.1.

Analysis of the summary brings out the fact that the four way match does a better job than the regular three way match except when the next two characters following the error match. In this case, after the three way match rules that the current character of the misspelled word is to be discarded (type *E* error, we have an extra character) or the current character of the word from the lexicon is to be inserted (type *D* error, we have a missing character), pure four way match has a potential inferior behavior.

Using this analysis, we act on the four way match only if the next two characters of the two words, right after the first error, do not match. In other words, the two methods must be mixed for the best result.

Table 5.6 Comparison of the Two Methods for Missing Characters

type	three way comparison	four way comparison
A	Only one missing: char(m) is conceded missed. Start next comparison at ‘n+2’ and ‘m+3’.	Substitute char(n) with char(m). Start next comparison at ‘n+2’ and ‘m+2’ which most likely do not match.
B	Missing: char(m). Reverse order char(n+1) and char(n+2). Start next comparison at ‘n+3’ and ‘m+4’.	Substitute char(n) with char(m). Start next comparison at ‘n+3’ and ‘m+3’.
D	Missing: char(m). Also, missing char(m+2). Start next comparison at ‘n+2’ and ‘m+4’.	Substitute char(n) with char(m). Start next comparison at ‘n+2’ and ‘m+2’.
E	Missing: char(m). Extra char(n+1). Start next comparison at ‘n+3’ and ‘m+3’.	Substitute char(n) with char(m). Start next comparison at ‘n+3’ and ‘m+3’.

Table 5.7 Comparison of the Two Methods For Added Characters

type	three way comparison	four way comparison
A	Extra char(n). Start next comparison at ‘n+3’ and ‘m+2’.	Substitute char(n) with char(m). Start next comparison at ‘n+2’ and ‘m+2’ which most likely will not match.
B	Extra char(n). Reverse order of char(n+2) and char(n+3). Start next comparison at ‘n+4’ and ‘m+3’.	Substitute char(n) with char(m). Start next comparison at ‘n+3’ and ‘m+3’.
D	Extra char(n). Also, missing char(m+1). Start next comparison at ‘n+3’ and ‘m+3.’	Substitute char(n) with char(m). Start next comparison at ‘n+2’ and ‘m+2’
E	Extra char(n). Extra char(n+2). Start next comparison at ‘n+4’ and ‘m+2’.	Substitute char(n) with char(m). Start next comparison at ‘n+2’ and ‘m+2’ which do not match.

5.5 Summary of Comparison Method

The algorithm for comparison of two words is an extension of the algorithm that is used in Lee [1990], which in turn was based on the research of Hall and Dowling [1980]. Here we summarize the extensions:

Table 0.1 Comparison of 3 Way Match with 4 Way Match

current error	next type of error	3 way comp. number of errors	4 way comp. number of errors	Algorithm which reports fewest extraneous errors
Missing character	**No error**	1	1 or 2	3 way?
	Reversed order	2	1	4 way
	Missing character	2	2	EQUAL
	Added character	2	1	4way
Added character	**No error**	1	1 or 2	3 way?
	Reversed order	2	1	4 way
	Missing character	2	1	4 way
	Added character	2	2	EQUAL

1. Extending the disagreement limit from 40% to 50% and lowering the limit dynamically as we find acceptable candidates.
2. Assigning error weights to different type of character comparisons so the result of comparisons can be ranked.
3. Adding the four way match to improve the comparison mechanism.

5.6 Spelling Correction Optimization Methods

There are two prerequisites for spelling correction:

1. A collection of words to compare the erroneous input with.
2. A function that compares the two words.

To get the best result from the spelling correction, the sure way is to compare the erroneous input word with all words in the lexicon to choose the best matching word. As the lexicon size grows, this method becomes impractical. We need to come up with a method to reduce the number of words from the lexicon for comparisons.

More often than not, the initial character of the misspelled word is not part of the error. I have exploited this characteristic by starting the search in the lexicon at the beginning of the words that have the same initial characters as the misspelled word. The lexicon is alphabetically sorted and is treated as a circular array. The comparison starts at the segment with the same initial characters as the erroneous input word. This effectively partitions the lexicon into 26 segments, each containing all the words beginning with a particular character, Figure 5.45. If we find acceptable candidate(s) within the first segment, we do not need to continue the search within other segments of the lexicon. If, however, we do not arrive at an acceptable word, our circular approach allows us to get to other subsets of the lexicon.

The segments do not have to be searched in sequence. After the first segment,

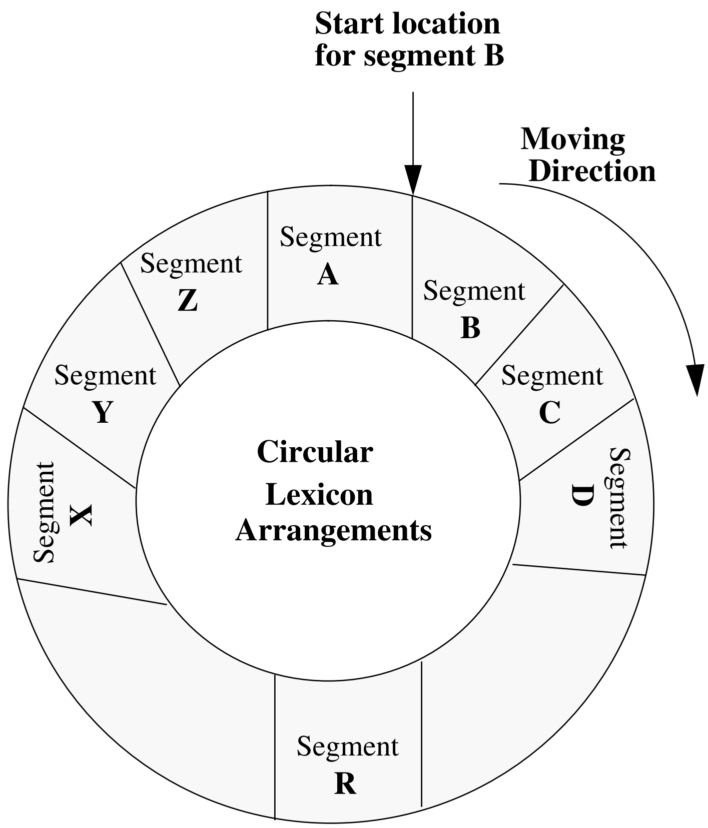


Figure 5.45 Segmented Lexicon

we could jump to another segment in which the initial character of the segment has a similar sound as the misspelled word. This may not be very useful in English as there are not many similar sounding characters in the alphabet. In languages like *Persian*, in which more characters sound the same, this solution can reduce the number of comparisons significantly[1]. In this way the segments are treated as a linked list, as opposed to a circular array. The segments get linked together dynamically depending on the initial character of the erroneous word.

This method could be extended to adjacent character misspelling of the first letter. After we fail in the segment with the same initial character as the erroneous word, the next segment is the one in which the initial character is adjacent to the first letter of the misspelled word. Of course, there is more than one letter adjacent to a key in the keyboard, and each must be searched one by one.

In the worst case, when the initial character is misspelled, the method behaves the same as other methods, except that the order in which the lexicon is searched is changed, with no penalty.

---

[1] For example, in *the Persian* language, there are four characters (ze, zal, zein, zat) that sound like ‘z’ as in *Zebra*. There are three characters (se, seen, sat) that sound like ‘s’ as in *Sea*. There are a few other letters in groups of two that sound similar.

The spelling error correction system presented here is not solely dependent on this issue. It only starts the search from the section of the lexicon that has the same initial character as the erroneous word. This should be considered as a guess. Having a good guess about where to start in the lexicon can potentially reduce a large search to a small one, since in most cases, when a close candidate is found, the search is terminated.

Summarizing the discussion, here are the details of the spelling correction algorithm:

max_limit = 50 X length of the misspelled word. This is the maximum disagreement threshold. No word would be accepted to replace the misspelled word if the disagreement reaches this number.

average_limit = 30 X length of the misspelled word. This is the average disagreement threshold. The disagreement limit starting at 'max_limit' is dynamically reduced as better replacement candidates are found. This is the minimum number that the limit gets reduced to, except when the spelling list is full, in which case, the disagreement limit is set to the maximum value of the disagreement in the list.

min_limit = 10 X length of the misspelled word. A good replacement candidate is a word with a disagreement with the misspelled word that is less than min_limit.

max_size = 10; This is the maximum number of replacement words in the spelling list. The spelling corrector keeps a list of replacement words. This number is the maximum size of the list.

The start location of lexicon is set to the segment with the same letter as the initial letter of the misspelled word

Loop until all words of the lexicon are considered, or an acceptable word is located.

Compare the next word of the lexicon with the erroneous input word. As the two words are compared character by character, we watch for the total value of the error weight. When the error weight is larger than max_limit, this step rejects the word from the lexicon as a replacement. Otherwise, the algorithm considers this word as a spelling error candidate and the following processes takes place:

1. If the only difference is the character case, then terminate the loop. Assume that this is the matching word. Needless to say, this test is done for the case in which the system is character case sensitive. Otherwise, to speed up the processing and avoid unnecessary comparisons, an equivalent word is built which is the same as the erroneous word, but all characters are in lower case. This works well for capitalization of words, i.e., the initial word of a sentence.
2. If the disagreements between the two words is less than min_limit of the word length, we accept the comparison word from the lexicon, and correct the erroneous word to it.
3. If there are only extra characters at the end of the erroneous input word,

this implies that a blank space might be needed (in other words, there is a possibility of two words being joined). If the extra characters of the erroneous input word match with a word from the lexicon, then terminate the loop, this is the matching word.

4. If there are only extra characters at the end of the word from the lexicon, this implies the possibility of an extra blank space (we may have a split word), or we may have an abbreviation. If the extra characters of the word from the lexicon match with the next input word following the erroneous input word, we consider the two input words as a split word. Otherwise, we have an abbreviation. In either case, the word from the lexicon is considered the replacement word for the misspelled word, and we terminate the loop.
5. If one of the matches is part of a phrase and the word before or after the erroneous word combined with the spelling choice makes up a phrase, then this is a good choice. This can be extended further, specially when the word by itself is meaningless. For instance, 'vice versa' is a meaningful phrase but neither of the words 'vice' and 'versa' have any meaning by themselves. If either one has a misspelled character, the phrase 'vice versa' is helpful to look at.
6. Add the word from the lexicon and the comparison result to the list of best matches. If the list is full, add this word only if its error weight is smaller than the largest error weight in the list. In which case, drop the last word in the list as this word is a better candidate.
7. If the list is full, lower max_limit to the largest error weight in the list. Otherwise, if the calculated error weight for the word from the lexicon is between the *max_limit* and the *average_limit*, then lower *max_limit* to the calculated error weight.

END LOOP.

If we terminate with any one of the conditions 1 to 4 of the above simplification criteria, then we have only one match. Otherwise, all matches must be considered.

An important simplification is to make the system to be character case ignorant. The spelling correction operates in two modes: (1) character case sensitive, (2) character case ignorant (both implemented). A graphical representation of this algorithm is presented in Figure 5.46.

The disagreement between the misspelled word and the word from the lexicon is shown in form of a ball. The bigger the disagreement, the larger the size of the ball. The ball is rolling down on a surface with three traps:

1. The smallest trap is *min_limit*.
2. The middle trap is *average_limit*.
3. The largest trap is *max_limit*.

As the ball rolls down it falls into the first trap that is larger than its size. The first trap is different from the last two. If the ball drops down in the first trap, the process is terminated. Each one of the other two traps has a trigger. If the ball falls in either one, the associated trigger is activated. Finally, the replacement words are collected in a container. Once the container is full, it drops the biggest size ball and the ball activates the third trigger. The three triggers perform the following actions:

1. Close the extended disagreement trap. Note that once we find a word that meets the average disagreement, there is no need for extended disagreement.
2. Lower extended disagreement to the disagreement of this word. We have found a word that has a disagreement less than the extended one, therefore, lower the extended disagreement to the size of the ball.
3. Lower average disagreement to the worst one in the container, and close the extended disagreement. This action forces us to only look for words that are at least better than the worst match that we have found so far.

5.7 Using Syntax to Improve Spelling Correction

In the previous section, we talked about the best match being the way humans correct spelling errors. In this section, we concentrate on help from the syntax. Obviously, the syntax of the language is useful for choosing the best candidate amongst a few possible matching words when there are different parts of speech amongst the candidates. Another helpful hint could be applying the semantic rules, like the tense of the verb with respect to the rest of the sentence. For instance, in the sentence:

the cake was baten

the spelling error algorithm could bring two candidates: ‘eaten’ and ‘beaten’. We know from semantics that we cannot beat a cake, and most likely we eat the cake. So by using the semantics, we prefer the candidate ‘eaten’ over ‘beaten’. Also, if there is a possibility of a phrase, then the best word for replacing the erroneous word is the one that helps construct the phrase. For instance, if we have ‘bice versa’, the word ‘bice’ matches with ‘rice’ and ‘vice’, but ‘vice’ completes the phrase ‘vice versa’ so it has priority over ‘rice’.

This approach is built on the concept that the parser is capable of handling a word with multiple parts of speech. When we have a misspelled word, all its replacement candidates for correction are passed to the parser as the next word from the lexicon. Note that if two candidates have the same part of speech, the syntax rules cannot help to distinguish between them, and, we must rely on the semantic rules to help us choose the better candidate. The idea is that not all of the replacement candidates are syntactically and semantically valid for the context of the sentence and this helps shrink the number of candidates, we hope, to one.

An example is provided to clarify the concept. For the input: “the boys **wer** lost”, our spelling corrector finds two equally weighted candidates:

1. “were”, ***e*** added at the end,
2. “we”, ***r*** removed at the end.

Both have a weight of 46. The parser, using the context of the sentence, selects “were” as we need an auxiliary verb.

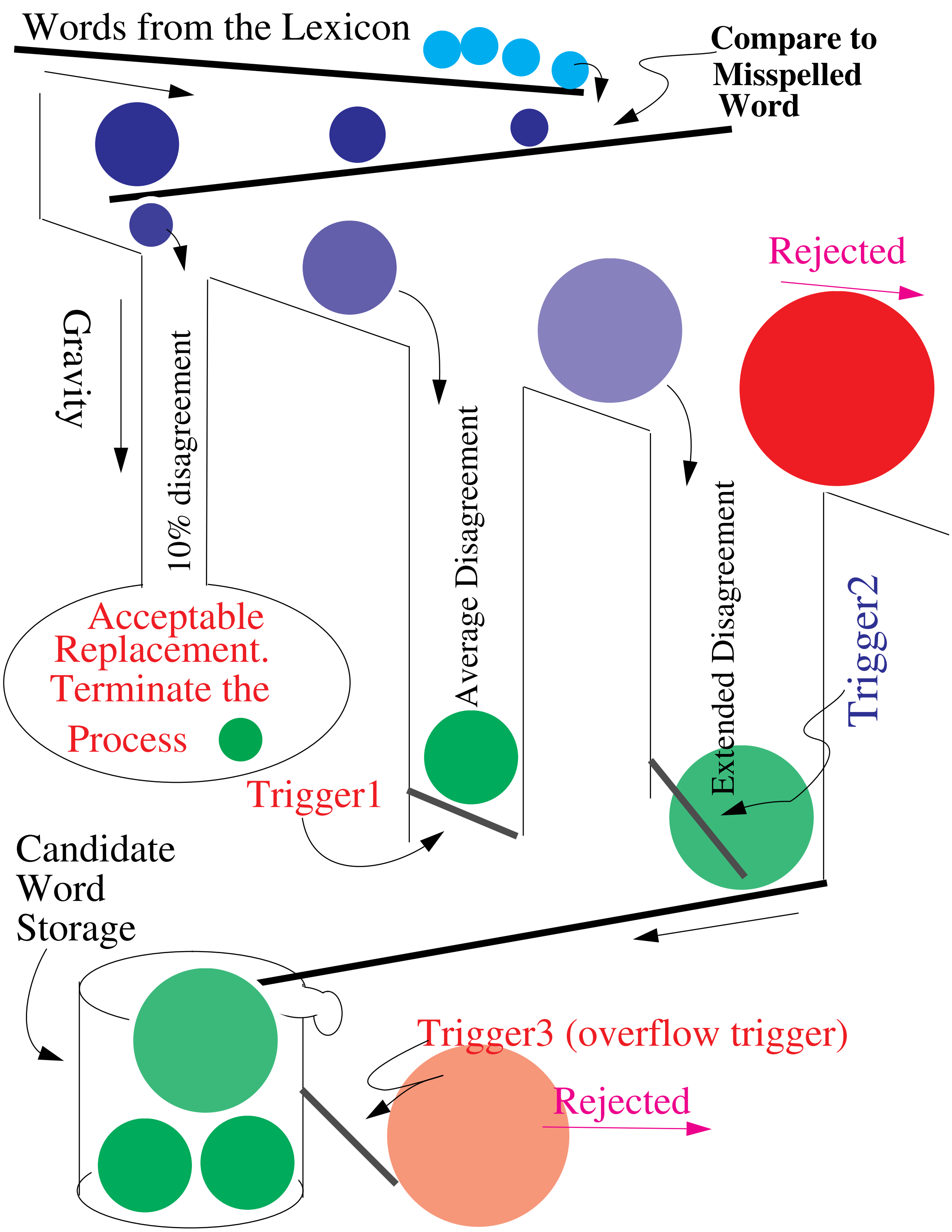


Figure 5.46 Graphical Representation of Comparison Algorithm

There are two ways that the parser and spelling corrector could cooperate:

1. Hints from the parser: As soon as an erroneous input word is encountered, the spelling corrector asks the parser to give a list of possible parts of

speech, pos, that are syntactically acceptable. Then, the spelling corrector only compares the words in the given category with the erroneous word. This method is used in the *NOMAD* system [Granger et al., 1983]. Hall and Dowling [1980, p. 397] use the notation $G = (V_N, V_T, P, S)$ for a grammar, V_N are the nonterminals, and V_T are the terminals, P the productions, and S the start symbol in V_N. In error-correcting parsing, instead of rejecting a string found to be in error during the parsing, the string is corrected to that member of $L(G)$, the language generated by the grammar G, which is closest to the given string.

2. The spelling corrector finds all possible matching candidates and passes them to the parser. Then the parser, in turn, selects syntactically valid candidates only.

Let us analyze the advantages and disadvantages of both methods.

With the first approach, with the parser giving hints to the spelling corrector, by considering the list of all possible candidates, the spelling corrector can eliminate a lot of entries. If the lexicon is not sorted by syntactic categories, the advantage of this method becomes less significant. There are two problems with this approach:

1. We are not in a single parse tree environment. We are dealing with naturally ambiguous grammars. The parser cannot give a complete list of syntactically expected word categories until all possible parse trees are built. Therefore we have a choice:
   1.1. Give the partial list. Generate the possible categories that the current tree can generate. Next time when we are in another branch we give some other list to the spelling corrector for searching for the candidates, effectively, doubling the work of the spelling corrector. Note that it does not stop here. As we get to the third parse tree, its list must be considered and so on.
   1.2. Stop the parser at this branch and go back to all other possible branches until all branches get to parse the current erroneous word. This solves the problem of the partial list but adds a lot of work to the parsing for suspending the work in progress.
2. The sentence is not necessarily syntactically correct. In other words, the spelling of the word is not the only possible thing that can go wrong. Let's say that the intended sentence is:

   I saw men ...

   but the user forgets to type the verb: 'saw' and also types 'men' as 'mex':

   I mex ...

   There are two errors here, one missing verb, and the second one is the misspelled word 'mex'. The parser expects a verb and we have a misspelled word which is suppose to be 'men' of category 'noun'. What does the parser do at this point? It certainly gives the verb category to the spelling corrector for consideration and, therefore, misleads the spelling corrector.

   Note that we do not have always a complete sentence. In a fragment, the parser can get really confused.

The only disadvantage of the second approach (having the spelling corrector find all the best matches regardless of their syntax category), is that the spelling corrector must look for all possible candidates, as opposed to the other method, which only looks for a particular part(s) of speech. We noted earlier that the hints from the parser can be misleading.

This second approach requires no change in the parser algorithm, as the parser is able to deal with words of multiple parts of speech. I use this approach and have the spelling corrector find all possible candidates.

If the parser is the only module to use the context for spelling correction with no semantic check, then the spelling corrector must give the best candidates from each syntax category to the parser. This is because the parser cannot distinguish amongst the words with the same syntax category.

In case there are semantic rules to help the spelling corrector, then all candidates must be included in the list of possible corrected candidates. The parser shortcoming can be overcome by use of semantic properties of the replacement words. The properties could be the tense of the verb, its being singular or plural, first and second person verses the third, or finally its gender, and other semantic properties such as case information.

The candidate words within the same part of speech are distinguished from each other during the process of building the Lexical Functional Grammar (LFG) f-structure. We only take the word that unifies with the rest of the input word and generate an f-structure that is complete and coherent. For example, in the following sentence:

“lohng sees Mary.”

There are two candidate words for the erroneous word “lohng”: “John” and “long”. The parser eliminates the verb, adjective and adverb senses of “long”. Therefore, the only sense of “long” the parser finds acceptable is the is “noun” sense. Now, both of these words are of type “noun”, so the parser cannot distinguish between them. Semantically, these two words are different, John can be the subject for the word sees which requires one, therefore we eliminate ‘long’ from consideration.

5.8 Summary

The spelling correction algorithm presented in this chapter does the following:

1. Extends the three way match algorithm to the four way match algorithm to achieve a more realistic comparison of two words and produce a better correction of the first word to get to the second word. Note that the four way match uses only two character look ahead as in the original algorithm, so it does not add complexity or extra overhead.
2. Assigns error weights for different types of corrections so there is a solid method to compare two or more replacement candidates for an erroneous word.
    2.1. Keyboard adjacency character mistype: if the replaced character is adjacent to the erroneous character, the error weight is lowered.

2.2. A missing silent character in an erroneous word has lower error weight than a regular character, provided that the lexicon contains the information about silent characters. For example: ‘sychology’ for ‘psychology’, missing ‘p’.

2.3. Substitute characters that sound similar have lower error weight. For instance, ‘c’ and ‘k’ as in ‘kool’ as ‘cool’.

3. Filtering capability of the spelling corrector to discard comparison of the misspelled word with irrelevant lexicon words. The filtering starts with a wide acceptance for words (the initial limit on the number of disagreements is set to 50%). As better candidates are found the filtering will be tighter (the disagreement limit is lowered to the worst of the best matches found so far). As the two words are compared character by character, the error weights are calculated. As soon as the accumulated error weight reaches the disagreement limit the process is terminated.
4. Phrase Consideration: If the erroneous word appears in a phrase existing in the input, then the word matching with it in the phrase will be the candidate, even if there is another word with lower error weight, which does not appear in the phrase.
5. Allows spelling corrector to return more than one replacement word for a misspelled word. The best replacement candidate is the one that does not violate syntax rules.
6. Uses the syntax and semantic rules to help the spelling correction.
7. In addition to using syntax for finding better candidates (or syntactically acceptable candidates), a semantic check is added to the process. This is done using the LFG f-structure construction algorithm, which distinguishes between two candidates with the same part of speech (which parser cannot distinguish).

The spelling correction process finds up to ten (a configurable number), closest candidates words for a misspelled word. The spelling corrector relies on the parser to apply syntactic and semantics rules to narrow the choice. This allows us to avoid asking users to intervene most of the time.

CHAPTER VI

LEXICAL STRUCTURE

6.1 Introduction

This chapter describes the layout of the lexical entries. The system structure was designed with the following objectives:

1. Fast access to lexical information: The time spent to extract the lexical information for an existing word in the lexicon should be minimal. In other words, the structure must support a fast access method.
2. The text of lexical entries must be available to the spelling corrector without any restrictions. The text must be alphabetically sorted to allow the spelling corrector to perform its task.
3. Abbreviated entries as well as phrases must be easily accessible to the lexical analyzer.
4. There should not be any restriction on how many parts of speech a word can have.
5. There should not be any restriction on how many *senses* a word can have. A word can have several *senses* for a single part of speech. As we saw in Section 2.4 the word *dish* has multiple senses as a noun as well as a sense as a verb.
6. There should not be any restrictions on how many semantic roles a word can have.
7. There should not be any restriction on the number of words in the lexicon. It must be extendable as [Kukich, 1992a, p. 83] mentions, because no lexicon will ever be complete since novel words will always arise.

The lexicon information includes the word syntactic category (part of speech) and list of LFG functional schemata as well as other information such as Case Frame information for verbs. For example the entry for one sense of the word ‘increases’ is shown in Figure 6.1.

The “word string” may be:

1. single word: i.e. ‘increase’ as a noun, or a verb.
2. phrase: i.e. ‘blood pressure’ as a noun.
3. abbreviated entry: e.g. ‘bp’ for ‘blood pressure’. Note that a word could have its own lexical entry and also be an abbreviation of another lexical entry. For instance, ‘i’ is a pronoun, and at the same time, is an abbreviation of the word ‘increase’.
4. compound entry, a single word that needs to be split into two words or more (word to be expanded), i.e. “cannot” for “can not” or “can’t” for “can not”. Entries like ‘its’ are more difficult, since it is both a simple possessive similar to the word ‘my’, and also the abbreviation of ‘it is’. (Note that we have to expect that users will drop apostrophes.)

and the lexicon information may be:

1. Part of speech.
2. LFG functional schemata.
3. Case frame for verbs.

Representation of item

Syntactic category

increases V $(\uparrow \text{PRED})=$ 'increase<($\uparrow$SUBJ) ($\uparrow$OBJ)> '
$(\uparrow \text{SUBJ NUM}) = \text{SING}$
$(\uparrow \text{SUBJ PERS}) = 3$
$(\uparrow \text{OBJ PAR}) =_c +$

List of functional schemata

Figure 6.1 Format of Lexical Items

6.2 Lexical Representation

The lexicon consists of a collection of lexical entries organized in a hash table. Many words have multiple parts of speech. Often a word has several *sens*es with the same part of speech. Important examples in our sublanguage are: *change*, *increase*, *decrease*. Note that increase, as a verb, means both "go up" and "cause to go up". These different senses have different arguments. A lexical entry is the full definition for one *sense* of the word. This is to say that if a word has multiple *senses*, then it has multiple lexical entries.

Note that the parser works with the part of speech. Different senses of a word with the same part of speech is considered during the construction of f-structure. With this in mind, lexical entries are compacted as follows:

1. There is one lexical entry for each part of speech of a word.
2. There is a list of semantic data for each pair: (word, pos). Each semantic data list pertains to one sense of the word.

Section 6.2.1 describes the handling of words with a single part of speech. Section 6.2.2 describes the handling of words with multiple parts of speech. Note that whether a word has a single part of speech or multiple ones, it can have multiple senses within each part of speech.

6.2.1 Lexical Entry with a Single Part of Speech. In the simplest form, a word has a single part of speech. There will be a single entry for these words. For example 'positive' has a single part of speech *adjective*, shown in Table 6.1.

Table 6.1 Lexical Entries for 'positive'

Lex Sym	Part of Speech	Semantic Data
positive	adjective	((PRED positive) (DELT +) (CHG +))

6.2.2 Lexical Entry with Multiple Parts of Speech. Words that have more than one part of speech, have one entry per part of speech in the lexicon. The lexicon information is saved in an array that is alphabetically sorted with respect to the text of the entries. Multiple entries of a single word are contiguous as all entries in the lexicon are sorted.

Each *sense* of a word may have a list of functional annotations. For example, the active form of "add" may or may not have a "to-value", it will have the following two alternative predicates:

$$(\uparrow PRED) = \text{'}add<SUBJ\ OBJ\ TO\text{-}PP>\text{'}$$
$$(\uparrow PRED) = \text{'}add<SUBJ\ OBJ\ NIL>\text{'}$$

Note that both of these PREDs occur for the same sense of "add", so we would prefer to put "add" in the lexicon only once with the list of alternative predicates attached. The parser should loop over the list of alternative predicates to find the acceptable one.

We could have repeated lexical entries, one for each version of "add." This approach will work for now, but this will definitely waste space. Beside, when we have multiple entries, each entry implies a different sense of the word. This can get worse, for example, "added" as a past participle occurs in a passive with two different (independent) deletions, and it can also occur in an active sentence with an auxiliary verb:

1. Blood was added by venous return to the atrium.
2. Blood was added by venous return.
3. Blood was added to the atrium.
4. Blood was added.
5. Venous return has added blood to the atrium.
6. Venous return has added blood.

Here *to-pp* and *by-pp* are present/absent in the passive form of add, and *to-pp* is present/absent in the active form.

As mentioned before, there are two senses for the word *increase* as a *verb*. When there is no object in the sentence with *increase* as the main verb (such as X increases) we have the structure shown in Figure 6.2. When we have an object in the sentence (such as X increases Y), then we have the structure shown in Figure 6.3.

increases V $(\uparrow \text{PRED})= \text{'increase}<(\uparrow \text{SUBJ})>\text{'}$
$(\uparrow \text{SUBJ NUM}) = \text{SING}$
$(\uparrow \text{SUBJ PERS}) = 3$
$(\uparrow \text{SUBJ PAR}) =_c +$

Figure 6.2 Lexical Function for Intransitive Sense of Increase

increases V (↑PRED)= 'increase<(↑SUBJ) (↑OBJ)> '
(↑SUBJ NUM) = SING
(↑SUBJ PERS) = 3
(↑OBJ PAR) = $_c$ +

Figure 6.3 Lexical Function for Transitive Sense of Increase

To save space in the system, there is one entry for *increase* as a verb in the lexicon. However, the entry has two semantic structures combined. This way the parser builds the same c-structure for both of them, but the LFG f-structure builder examines the tree, and when there is no object present, it selects the structure shown in Figure 6.2. If there is an object present, then the verb increase must mean 'cause to go up' and therefore we select the second sense of the verb *increase*. These two structures are combined as:

```
((OR
        ((PRED "increase2" SUBJ OBJ)
          (SUBJ ((NUM SING)
                 (PERS 3)))
          (OBJ ((check+ PAR +))))
        ((PRED "increase1" SUBJ)
          (SUBJ ((check+ PAR +)
                 (PERS 3)
                 (NUM SING)))))

(TENSE PRES))
```

Note except the common attribute value pair: '(TENSE PRES)' we have two separate list of functions for the verb *increase*.

It is worth mentioning that there is another lexical entry for *increase* as a *noun*. It happens that *increase* as a noun has two senses:

1. as a nominalization of the intransitive verb (*x increases*), for instance:

   the increase in venous return

   or

   the increase in price

2. same meaning, but the subject is understood from the context, as in:

   When somebody talks about trading stock options, I never understand whether it is the ***increase*** *or the decrease that makes the profit.*

The second sense of *increase* is syntactically different, even though it has the same meaning, so we must make a second lexical entry for it.

6.2.3 Compound Lexical Entry. In order to normalize all parts of speech and all possible combinations, we will have a compound lexical entry for words that are combinations of other words. The structure for the type compound is a list of the words that make up the compound word. Table 6.2 shows a few lexical entries for compound words.

Table 6.2 Sample Lexical Entries for Compound Words

Lex Sym	lexical entry
cannot	(Compound Defstructure: can not)
can't	(Compound Defstructure: can not)
it's	(Compound Defstructure: it is)

6.3 Lexical Collections

All lexical entries are put together in an alphabetically sorted array. Note that the lexical entries for a word with multiple parts of speech follow each other sequentially. When we access the lexicon from outside, if we have the first index of a word entry in the lexicon, we can access all of its entries in the lexicon. For instance, all entries for the word 'increase' are contiguous. Table 6.3 shows a sample lexicon.

6.4 Accessing the Lexicon

Once the lexicon is built, there is a need to have an efficient access method. The main usage of the lexicon is to get the lexical information about a given word or a phrase. In a simple term, we could consider the lexicon as a database of word properties. On one hand, it should allow insertion of new words (as well as modifications of the existing words). On the other hand, given a word, it should provide the properties of the word (lexical entries).

The basic retrieval requirement is that a string of characters: 'str' are passed to the lexical analyzer, and it returns the lexical properties of 'str'. The given string: 'str' can be a single word, an abbreviation of a word or a phrase, or it can be a phrase (multiple words).

The parser and the text generator access the lexicon in slightly different manners. The text generator requires basic access to the lexicon as it starts from a string and requires a lexical entry.

The parser has greater needs as far as accessing requirements. In addition to the basic retrieval facilities, the parser needs support for spelling correction and phrase recognition. Note that the text generator accesses the lexical entries for a phrase by submitting the full phrase, not one word at a time. The parser, on the other hand, requires construction of phrases as the input words are fetched one at a time.

The parser depends on the lexical analyzer to get the part of speech of the next input word. As the lexical analyzer reads the next input word, the method defined here is used to give the following information to the parser:

1. Part of speech as well as other lexical information for the word, if there is any.
2. If the word is an abbreviation, the parser and the generator must be notified that the word is an abbreviation, and also it must be given access to the full

Table 6.3 A Sample Lexicon

index	word entries	lexical entry
1	addition	<lexical entry for ‘ *addition* ’>
2	blood	<lexical entry for ‘ *blood* ’>
3	blood pressure	<lexical entry for ‘ *blood pressure* ’>
4	blood volume	<lexical entry for ‘ *blood volume* ’>
5	bp	<Abbreviation for ‘ *blood pressure* ’>
6	bv	<Abbreviation for ‘ *blood volume* ’>
7	can’t	<must be de-abbreviated to can not>
8	cannot	<must be de-abbreviated to can not>
9	go	<lexical entry for ‘ *go*’>
10	i	<lexical entry for pronoun ‘ *i*’>
11	i	<Abbreviation for ‘ *increase* ’>
12	in	<lexical entry for ‘ *in* ’>
13	in addition	<lexical entry for ‘ *in addition* ’>
14	in addition to	<lexical entry for ‘ *in addition to* ’>
15	increase	<lexical entry for noun ‘ *increase* ’>
16	increase	<lexical entry for verb ‘ *increase* ’>
17	it’s	<must be de-abbreviated to ‘ *it is* ’>
18	its	<lexical entry for possessive ‘ *its* ’>
19	president	<lexical entry for ‘ *president* ’>
20	pressure	<lexical entry for ‘ *pressure* ’>
21	saw	<lexical entry for noun ‘ *saw* ’>
22	saw	<lexical entry for verb ‘ *saw* ’>
23	to	<lexical entry for ‘ *to*’>
24	vice president	<lexical entry for ‘ *vice president* ’>
25	vice versa	<lexical entry for ‘ *vice versa* ’>
26	volume	<lexical entry for ‘ *volume* ’>

word.

3. If the input word is part of a phrase, the lexical analyzer must be prepared to look for a phrase. For instance, when the lexical analyzer reads in the input word ‘in’ it is notified that this might be part of a phrase, as well as a simple word. As the lexical analyzer continues and reads the next word, if

it happens to be 'addition' the lexical analyzer again is notified. Finally if the third word is 'to' the lexical analyzer must be notified that it is the phrase: 'in addition to', so then all three words make up one lexical entry of type *preposition*. If the third word is not 'to', the lexical analyzer concludes that it has the phrase: 'in addition', an *adverb*.

Providing for the above requirements, we put all words, phrases and abbreviations together in a structure and call the result the "word table". Another table, the "lexicon table" contains the essential information about the syntax (part of speech) and semantic data.

6.5 Word Table

The Word Table contains all the words and phrases that are in the lexicon. Note that a word may have multiple entries in the lexicon table (at least one for each part of speech), but it has only one entry in the word table. The main reason for the existence of the word table is to facilitate recognition of abbreviated words or phrases. All single words, including the meaningless ones like *'versa'*, are in the word table. Note that '*vice versa*' is in the lexicon, but there is no entry in the lexicon for '*versa*'. All phrases and abbreviations are also in the word table. There are no repeated word entries in this table unlike the lexicon table, which has as many entries for a word as there are part of speeches for the word. The word table has an index to the first and last occurrence of each word in the lexicon. The single entry for each word makes this table usable by the spelling corrector.

An entry in the word table may be in one or more of the following categories:

1. Simple entry: a single word with a single sense in our sub-language. There is an entry for it in the lexicon. The entry in the lexicon contains all information about this word, for example "go" or "pressure".
2. Abbreviation: an abbreviated word can simply stand for:
   - 2.1. another single word, like 'i' for increase.
   - 2.2. a phrase, like *bp* for blood pressure.
3. Phrases: A phrase consists of multiple words that all together have one lexical entry, one part of speech.
4. Compound entry: Same as a simple entry. The only difference is although it is a single word, it consists of more than one lexical entry.

Compound entries are exactly the opposite of phrasal entries. In a phrase, there are a few words that have one lexical entry. On the other hand, a compound entry is a single word that has multiple lexical entries, as it consists of multiple words.

A single word can be a compound entry as well as a simple entry, or an abbreviation, i.e. 'its' is a simple entry when it is part of a possessive phrase. It is also a compound of the two words 'it' and 'is'. This should not be confused with the phrases and abbreviations as they have a single part of speech, e.g., 'i' for 'increase' has one part of speech, whereas "its" could be a compound word: 'it' and 'is' in which each one has its own part of speech. Handling a compound word when it is mixed with a single entry is difficult. For instance, if the next word is 'cannot', the lexical analyzer returns two entries: 'can' and 'not'. Now, if the next word is 'its', should it return a possessive 'its' (a single word) or should it return two entries "it" and "is"? If the lexical analyzer returns both cases, (*its* , and *it is*) then input word numbers will be mixed, do we count it as one or two?

Note that a word could belong to more than one of the above categories. For instance, *I* is a simple entry, a *pronoun* , and also is an abbreviation for *increase* , and *bp* is an abbreviation for the phrase *blood pressure* .

The word table contains the list of words with the following attributes:

1. Word strings: character string, text of the word(s)
2. Index to the Lexicon: If the word has a defined structure in the lexicon, whether simple or complex, there is an entry here for it. If this attribute is missing, there is no defined structure for this word in the lexicon. *bp* has no entry here since by itself it is not a complete word. When there is a single number in this field it indicates that there is one part of speech for this word and the number is the index to the lexicon table. When there are two numbers in this field, we have a word with multiple senses. The first number is the index to its first entry in the lexicon and the last number is the index to its last entry in the lexicon. The range of entries between the first and the last number must be considered for this word.
3. Part of a phrase : This is a list of indexes to the phrases that this word is part of. Note that the entries for 'in', 'addition' and 'to' all have values in this field. Note that a necessary condition for a phrase is that it consists of more than one word. If there is no entry in this field, that indicates that the word is not part of any phrase in the lexicon. When there is an entry here, the lexical analyzer must watch for a phrasal entry. The actual phrase will be looked up in the hash table. The indexes are to the word table itself.
4. Abbreviated : This is an index to the word that this entry is an abbreviation of. The index is to the word table itself. The entry could be a phrase, e.g. '*bp*' is the abbreviation for the phrase 'blood pressure', or simply another single word like 'i' for *increase* . If it turns out that a single word can be an abbreviation of multiple words, we could make this field a list of indexes, like the list of phrases in the previous field.
5. Compound words: This slot contains a list of words to be substituted for the compound entry. For instance, the compound entry 'cannot' must be replaced with two words: 'can not'.

Table 6.4 shows the content of the word table for the sample lexicon of Table 6.5.

The three entries: phrase index, abbreviated index, and compound words are very sparse. The lex-builder program gives a choice to the user to generate the full table, or create a sparse matrix for each one of these three fields.

6.6 Lexicon Table

The Lexicon Table consists of the syntactic information, the part of speech, as well as the semantic information. The semantic information includes the LFG f-structure functional schemata [Wescoat 1987, p 6] entries for all words as well as

Table 6.4 Word Table

index	Word strings	lexicon index	phrase index	abbrev. index	compound words
1	addition	1	1002, 1003		
2	blood	2	1000, 1001		
3	bp			1000	
4	bv			1001	
5	can’t				can, not
6	cannot				can, not
7	go	5			
8	i	6		10	
9	in	7	1002, 1003		
10	increase	10, 11			
11	it’s				it, is
12	its	12			
13	president	13	1004		
14	pressure	14	1000		
15	saw	15, 16			
16	to	17	1003		
17	versa		1005		
18	vice		1004, 1005		
19	volume	20	1001		
	phrasal entries		***phrasal entries***		***phrasal entries***
1000	blood pressure	3			
1001	blood volume	4			
1002	in addition	8			
1003	in addition to	9			
1004	vice president	18			
1005	vice versa	19			

‘case frame’ data for verbs.

There are no entries for abbreviations since the lexical analyzer expands them using the Word Table. Note that there is no need to insert anything that is not semantically useful here. For instance, the word ‘versa’ by itself is meaningless, therefore there is no need for any entry in the lexicon. These types of entries are in the word table and the hash table for the lexical analyzer and the spelling corrector. The content of the lexicon table for our sample lexicon is shown in Table 6.5.

Table 6.5 Lexicon Table

index	word entries	lexical entry
1	addition	<lexical entry for '*addition*'>
2	blood	<lexical entry for '*blood*'>
3	blood pressure	<lexical entry for '*blood pressure*'>
4	blood volume	<lexical entry for '*blood volume*'>
5	go	<lexical entry for '*go*'>
6	i	<lexical entry for pronoun '*i*'>
7	in	<lexical entry for '*in*'>
8	in addition	<lexical entry for '*in addition*'>
9	in addition to	<lexical entry for '*in addition to*'>
10	increase	<lexical entry for noun '*increase*'>
11	increase	<lexical entry for verb '*increase*'>
12	its	<lexical entry for possessive '*its*'>
13	president	<lexical entry for '*president*'>
14	pressure	<lexical entry for '*pressure*'>
15	saw	<lexical entry for noun '*saw*'>
16	saw	<lexical entry for verb '*saw*'>
17	to	<lexical entry for '*to*'>
18	vice president	<lexical entry for '*vice president*'>
19	vice versa	<lexical entry for '*vice versa*'>
20	volume	<lexical entry for '*volume*'>

6.7 Lexicon Access

In the second version of Circsim-Tutor [Kim et al., 1989], the lexicon was organized in a trie structure [Lee, 1990]. Lee uses a three level trie. His approach gets us, very fast, to the collection of words that have the same three initial characters as the word being searched for. Considering that on the average, in a large lexicon, most words are longer than three characters we are faced with another search problem: what to do with the list of words with the same initial *n* characters? Figure 2.5 (repeated here for convenience: Figure 6.4) shows the word lengths of the Circsim-Tutor Lexicon (version 2). The average word size is 7.21.

In Section 2.3.4 on page 29, we saw that on the average there are six word comparisons to get to a word in the lexicon. In comparison, the hash table search requires 1.4 word comparisons to get to a word in the lexicon, see Section 6.10.1 on page 137.

Lee [1990] states that his trie takes over 18,955 words, for a lexicon of 1567 words. Each branch node contains 27 linked fields, 26 letters of the alphabet and a blank. At level 1 the branch node contains 26 letters of the alphabet and a special character. At any level all key values are partitioned into 27 disjoint classes depending on their first character. So, there exist a total of ((27 * 27) * 26 + 1 = 18,955) lists.

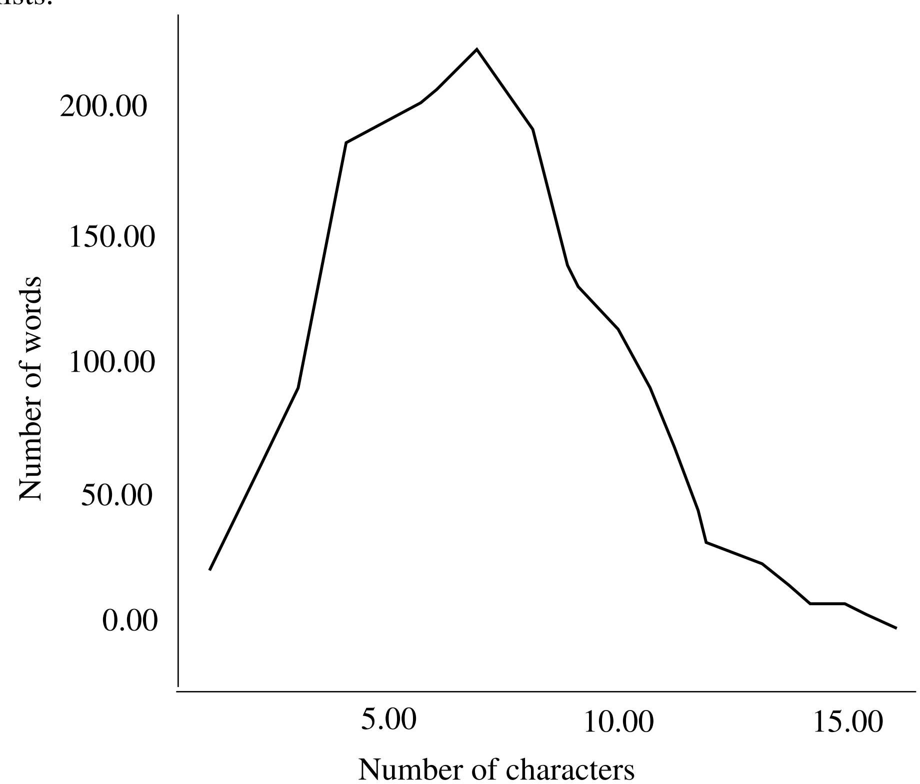


Figure 6.4 Circsim-Tutor Lexicon Word Length

Access to the lexicon, anticipating a large number of entries, is via a hash table [Knuth, 1973]. A good hash table organization with a good hash function provides a better and faster search as well as saving a lot of memory compared to a trie, as most of the entries in the trie will be blank. For instance, we saw that 18,955 lists were used in the trie.

In comparison with 18,955 lists used in the trie to hold the Circsim-Tutor version two lexicon (of 1567 words) , my hash table size has only 1949 words as the overhead, see Section 6.10.1 on page 137.

A detailed example is presented in later sections of this chapter, which gives a timing report for various hash organizations.

6.7.1 Hash Function. It is ideal to have a hash function that does not generate the same number for two or more words and which therefore produces no collision. In practice this requirement is relaxed for two reasons:

1. It is difficult to come up with a perfect hash. However, this can be done since all the words in the lexicon are known in advance. If we use a hash that is not perfect, then increasing the size of the hash table always makes the hash function generate better indexes (fewer collisions) with a tradeoff of space overhead for a large lexicon.
2. Efficiency of the hash function is an important issue: if it takes more time to generate a perfect hash index than to solve one or two collisions, obviously in practice we favor the more efficient hash function. Note that here we are using the hash table for efficient searching. Our goal is not to come up with a perfect hash function. Rather, we need an efficient search process that is the combination of the hash function and possible collision handling. Our hash function, and the method used to build the hash table follows the following basic principles outlined in [Fischer and LeBlanc 1988, p. 257]:

- hash function: h (n) depends solely on n.
- h can be computed quickly.
- h is *uniform* and *randomizing* in mapping names to hash addresses. That is, all hash addresses are mapped to with equal probability, and similar names do not cluster to the same hash address.

6.7.2 Hash Table Construction. Commonly the hash tables are constructed with chaining for collision resolution. In this method, all the elements that hash to the same slot are put in a linked list.

Collision resolution by chaining performance depends on the load factor. Given a hash table *T* with *m* slots that stores *n* elements, <u>load factor</u> α for *T* is defined as *n/m*, that is, the average number of elements stored in a chain [Cormen et al., 1994, p.224]. Note that in collision resolution by linked list chaining, the load factor could be less than, equal to, or greater than 1. The worst case behavior of hashing with chaining is when all elements hash to the same slot (given a hash function that always returns a constant). Cormen et al. [1994, p.225] state that a successful search takes time $\theta(1+\alpha)$, on the average, under the assumption of simple uniform hashing.

I used chaining to resolve collision for the lexicon hash table with a modified chaining mechanism. I avoided using a linked list for chaining for the collision resolution. Instead, I used the unused hash table slots for this purpose. When there is a collision, an available hash slot that is never used by any of the lexical words is chosen to load the new entry. I constructed the hash table by using an array that is larger than the size of the lexicon. I stored the linked lists for chaining inside the hash table, in the otherwise unused hash table slots.

This should not be mistaken for open addressing, in which all elements are stored in the hash table itself and pointers are avoided altogether. Instead of following pointers, in open addressing, one needs to compute the sequence of slots to be examined. For a detailed analysis of open addressing see [Cormen et al., 1994].

In my method obviously the load factor α is less than one, as the hash table is larger than the size of the lexicon. In linked list chaining most of the space is used in the linked list, whereas here the hash table is the only table that we are working with. The larger the hash table the less chance of collision (depending on the hash function). As the load factor lowers, therefore, the search time: $\theta\,(1+\alpha)$ will be smaller. Normally 20% to 30% increase in the size is sufficient, with the final size being a prime number. For a 25% increase in the size, the load factor will be: n / (n+0.25n) = 0.8. Thus the average search time will be on the order of 1+0.8=1.8 searches for a word search. In later sections, we will see that I have reduced the search time to 1.4.

A hash entry contains two slots, the data slot and the link slot. The link is used for collision resolution. To have an entry in the hash for the string s, the hash function is applied to s, which generates an index i. If the hash table position at location i is empty, insert the string s at location i and set its link to nil. Otherwise:

1. Compare the data in this location with s, if it is the same, we already have it, otherwise
2. If the link is null find the first empty slot in the hash table and insert s there, save the index for the new location in the link portion of the current slot (i). Otherwise,
3. Read the link value and use it as the new index and repeat from step (1).

Note that step (2) above inserts s at the empty space in the hash that could be a hash index for another string, therefore increasing the chance of collision. To avoid this problem, I use a temporary table for step (2) above. To speed up the task, the temporary table is actually an extension of the hash table itself. So instead of the hash table being of size m, which is its final size, we use a table of size $n{+}m$ for building the hash table, where n is the size of the lexicon minus one (worst case of a bad hash function that generates the same number for everything.) The hash function still generates indexes from 0 to m. Step (2) of the above algorithm writes s in the next available hash table starting at location $m{+}1$. Once the whole lexicon is constructed, then all the entries in the hash table starting at location $m{+}1$ are moved to the empty slots of the hash tables between locations zero and m, and, of course, the corresponding links are all updated. Finally, we save the first m slots of the table as the hash table.

The lexicon is built in two steps. The first step is creation or modification of the raw lexical entry file. The second step is the construction of the hash table and the rest of the tables. At this step, the entries are static, as we know what is in the lexicon. The hash architecture was chosen for efficient search operation, and we were not concerned with insertion efficiency, as it is done once (when the lexicon is being built).

In summary, the hash organization as shown in Figure 6.5 is as follows:

- Section A, from 0 to a, is the original size of the lexicon (the number of words with entries in the lexicon).
- Section B, from a to b, is the additional area for the hash, around 20% of *the size A*. Note that b must be a prime number. The hash table is the combination of these two regions ($A{+}B$). There is no distinction between

the A and B region. We have split them up here to emphasize that there is additional space for the hash table.

- Section C is the auxiliary area used for building the hash. The final hash table will be section A and B. Region C is the size of the lexicon, so our algorithm would work even if the hash function generates a constant number for all strings.

The hash function generates indexes 0 to $b-1$ for any string. If the hash function generates h for string s, and location h is empty then s is saved there, otherwise s is saved in the next available spot in region C. At the end all items in region C are put back in available areas in A or B.

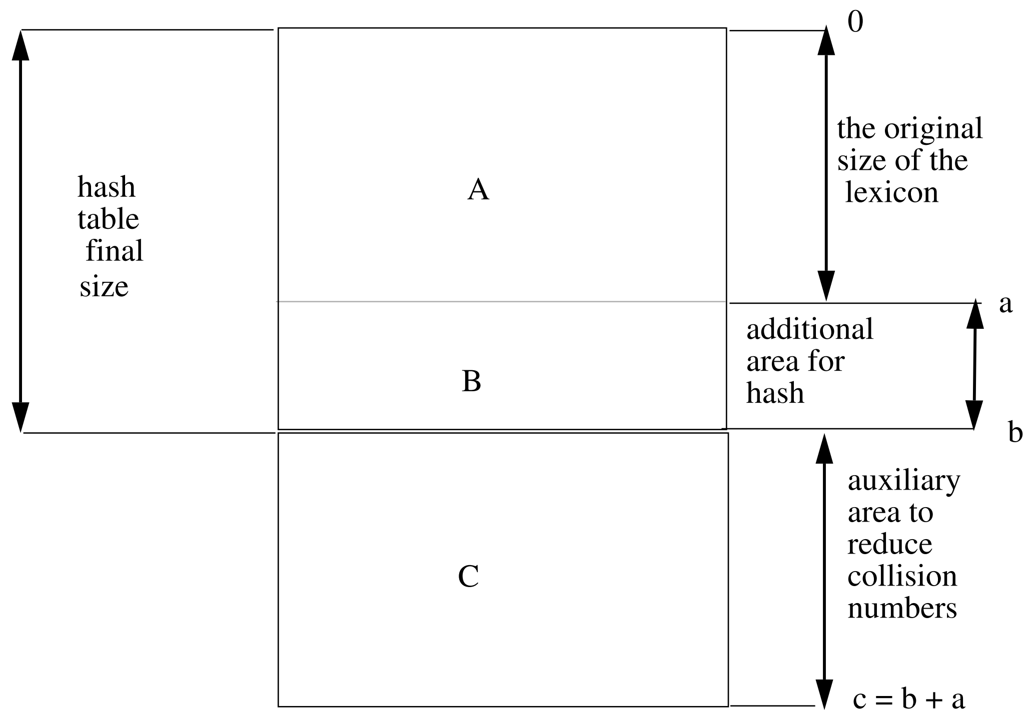


Figure 6.5 Hash Table Organization during Construction

This method improves the performance of the hash table in the following aspects:

1. The load factor is smaller than one. The space that is potentially used by the linked list is used as part of the hash table. This lowers the load factor without wasting space. The smaller load factor α, the lower the search time, as we saw : $\theta\,(1+\alpha)$
2. Since the data in region C (all collided entries) is outside of the hash table, while constructing the table, none of the cell spots, i, in the hash table were occupied by a string s such that its hash function value is different from i. So there is no chance of occupying the spot i and later another string with the hash function i will have to be added to the chain of string s.

Elements of Region C are moved to unused hash table slots. This would not create any problem for the existing words of the lexicon, as they would never hash to any of the slots used by region C. Misspelled words are different story. For one thing, unlike the existing lexicon words that are static, they are very dynamic because we cannot even guess what they will be until we see them (when we fail to find them in the lexicon). All we know about them is that we know what they are not (none of the words in the lexicon). Thus we do not know what hash value will be generated for them. If v is the value generated by the hash function for a misspelled word w, there are three possibilities:

1. v is an empty slot in the hash table.
2. v is the same as one or more hash value of the existing lexicon words. That is, there is one or more existing words in the lexicon, for which the hash function generates the value v.
3. v is different than the value generated by the hash function for any existing lexicon words, and is not the index of an empty slot (covered in the first case).

The last condition is the only one in which moving region C to the hash table (region A+B) causes additional collisions (only for none-existing words in the lexicon). In this case, the slot v is occupied by a word from region C, and we need to follow its link all the way to the end to realize that we do not have the word in the hash table. (Note that w was a nonexistent word in the lexicon.)

As mentioned above, the hash function generates indexes *between 0* and *b-1* for any string. I use a division method. That is, the hash function maps a key k into one of the m slots by taking the remainder of k divided by m:

$$h(k) = k \bmod m$$

The key k for a string s is calculated by successive addition of characters in s, and shifting the result. Section 6.10.1 provides a sample hash function. Good values for m are primes not too close to exact powers of two [Cormen et al., 1994, p. 228].

The next section provides details of a few experiments with building a hash table for the Circsim-Tutor lexicon as well as the Unix Dictionary. Section 6.10.1.3 gives details of experimenting with a simple hash function that avoids the division operation.

6.8 Lexicon Builder

To make it easy to add lexical entries, I have implemented a 'C' program, Lex_builder, that takes lexical entries as an input and generates the lexicon table, the word table and the hash table.

The lex-builder operates in two steps. The first step takes the raw lexical entries in the form:

a (pos art definite no))
a-m orientation (pos noun abbrev-for "actin-myosin orientation")
abdomen (pos noun cm count num sing gender neuter restriction cv-bpart)

and generates the lexical collections in the form shown in Table 6.3. Recall that the lexical collections are in the form:

"word string", "part of speech", "semantic structure"

For the above examples, the following entries are created:

“a”, “art” “((definite no))”
“a-m orientation” “abbreviated” “actin-myosin orientation”
“abdomen” “noun” “((cm count) (num sing) (gender neuter) (restriction cv-bpart))”

Note for the sake of simplicity, I used association list for the semantics entries. Actually, every semantic entry is a Lisp structure (destruct). Each part of speech has its own structure. The common afield amongst all of them is the “lex_function” field which contains the actual LFG annotation for the word.

The second step uses the lexical collections, generated in the first step, and creates the hash table, the word table, and the lexicon table.

Organizing the lex-builder to operate in two steps allows manual editing of the lexical collections.

To allow incremental addition of words to the lexicon, the system allows the words to enter the lexicon with or without part of speech. Further, the semantic structure (lexical function) for the word can be left blank. The system inserts default lexical functions at run time for the entries that have no lexical function.

6.9 Example: Accessing the Lexicon

We saw earlier that the word *increase* has two senses as a verb (*increase1* , *increase2*) and two senses as a noun (*increase3* , *increase4*). When the next input word is *increase* , the following steps are taken to get to semantic structures of the word *increase* :

1. Hash table is accessed to get the word table index.
2. Word table provides two indexes to the lexicon table, one for *increase* as a noun, and one for *increase* as a verb.
3. Each entry in the lexicon table has two semantic structures for each sense of the word *increase* .

Figure 6.6 shows a graphical view of accessing the word increase. It is interesting to note that the word “i” is a pronoun by itself and it is also an abbreviation for ‘increase’. If we search the lexicon for ‘i’, in addition to the four semantic data items for increase, we get one for the pronoun ‘i’.

6.10 Example: Building a Sample Lexicon

To test the efficiency of the lexicon, I first imported the Circsim-Tutor lexicon. It has 1,552 words. Later, I added the Unix dictionary, 25,146 words. I wrote a test function that basically loops through the lexicon and gets the hash index for each word. Then it goes to the lexicon and makes sure the word at the index is the same as the original word. (This also checks the reliability of the system by double-checking the hash indexes.) The function also takes an integer argument that indicates the number of iterations, i.e., if the function is called with 10 it performs the above task 10 times. To test the performance of the system for spelling errors, words that do not exist in the lexicon, another test was done by taking each string and adding the character ‘1’ to its beginning, this made the search fail. The times are given in seconds unless otherwise indicated.

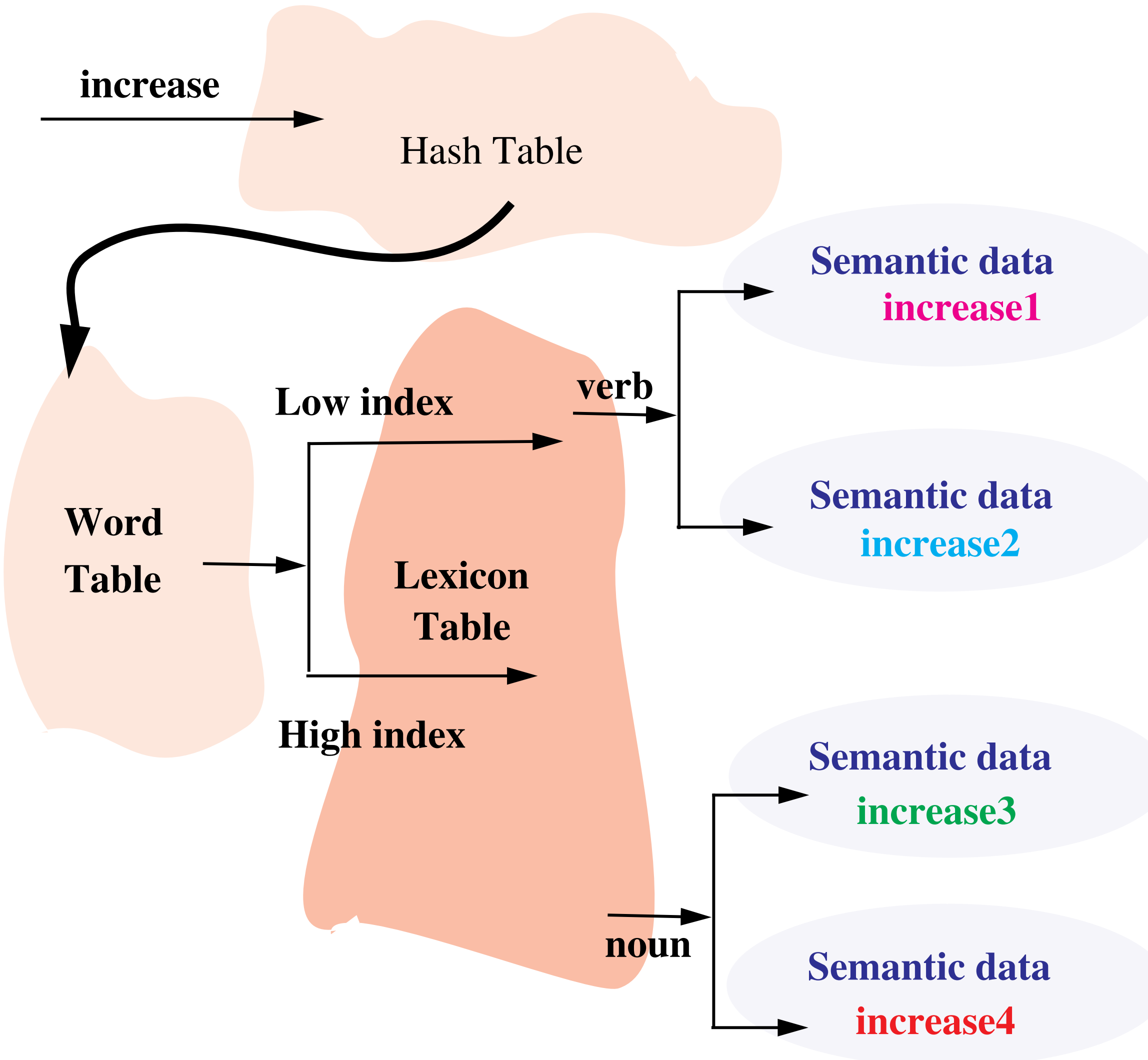


Figure 6.6 Accessing the Word Increase

6.10.1 Building the Lexicon in C. In C, in addition to building the hash table, I also wrote search algorithms using both binary search and also indexed binary search. In indexed binary search there is another table that gives the upper/lower bound of indexes in the lexicon for the initial character, so effectively segmenting the lexicon. Although binary search has a complexity of $\log_2(n)$, the improvement in performance of indexed binary search, compared to the binary search, is noticeable; especially for the second part of the test that searches for nonexistent words.

The hash function used is:

```
int hashFunc (char *str)
{
        unsigned int tmp = *str++;
        do {
            tmp ^=(tmp << 3) + (int) *str;
        } while (*str++);
        return (tmp % hash_entries);
```

}/* hash */

6.10.1.1 Filling 80% of the Hash Table. We start the test with a hash table of size 1,949 for the lexical entries of 1,552 words. The detailed report of hash construction is summarized in Table 6.6. The column “collision count” gives the number of collisions, and “number of entries” is the number of words for that collision count. The last column: ‘Percentage’ shows the percentage of the entries compared to the number of lexical entries.

Table 6.6 Collision Report for 1552 Words with Hash Size 1949

Collision count	Number of entries	Percentage
0	1066	68.6
1	370	23.8
2	97	6.25
3	15	0.96
4	3	0.19
5	1	0.06

In order to get to a word, there is one comparison for words with zero collisions, 2 for words with one collision, and 6 for words with 6 collisions:

$$1066 \times 1 + 370 \times 2 + 97 \times 3 + 15 \times 4 + 3 \times 5 + 1 \times 6 = 2178$$

Therefore, on the average, for existing words in the lexicon we have 2178/1552 = 1.40 comparisons. Compare this number to 6 comparisons for the Trie, see Section 2.3.4 on page 29.

In Table 6.7 we have two entries per search: hash, binary search, and indexed binary search. The first one is the time spent to search for a real word in the lexicon and the second number is the time spent to search for words in the lexicon that have an additional ‘1’ in the initial place, guaranteed to fail.

Table 6.7 Sample Lexicon with 1552 Items in C

No. of iteration	Hash table size	CPU time for Hash	CPU time for binary search	CPU time for indexed binary search
1	1949	0.099/0.133	0.183/0.223	0.099/0.116
10	1949	0.483/0.583	1.149/1.233	0.549/0.733
100	1949	4.399/5.416	9.916/11.282	5.116/6.583

6.10.1.2 Doubling the Hash Table Size. To see the effect of the hash table size, we doubled the hash table size (to the nearest prime number: 3,881, for 1,552 words). We keep the same hash function. The largest string of collisions is of size 5. Details of the collisions are shown in Table 6.8.

Table 6.8 Collision Report for 1552 Words with Hash Size 3881

Collision count	Number of entries	Percentage
0	1233	79.4
1	254	16.3
2	46	2.9
3	13	0.83
4	5	0.32
5	1	0.06

In Table 6.9 we have the result of the hash function but not binary search and indexed binary search as these two remain the same. The first number in column three is the time spent to search for a real word in the lexicon and the second number is the time spent to search for words in the lexicon that have an additional '1' in the initial place, guaranteed to fail.

Table 6.9 Sample Lexicon with Increased Hash Table Size

No. of Iterations	Hash table size	CPU time for Hash
1	3881	0.099/0.099
10	3881	0.516/0.599
100	3881	4.899/5.349

6.10.1.3 Simplifying the Hash Function. The hash function given in Section 6.10.1 generates the index by shifting the characters of the word and performs an *exclusive OR* operation. At the end of the hash function calculation, the index is divided by the hash table size, and its remainder is returned. Among other things, this operation guarantees that the generated hash index is a valid index to the hash table. The division operation is the reason for requiring a prime number for the hash table size.

In a special case, where the number of words in the lexicon is close (but smaller) than a power of two, the hash function can be simplified, by removing the remainder operation. This is what is needed:

1. Set the size as the next power of two: $n = 2^k$.
2. Since n is a power of two, therefore, the binary representation of the number $\underline{n-1}$ will be $n-1$ bits set to one, and the rest of the word is a set of zero bits:

   $$\begin{matrix} 0 \ldots 0 & 1 \ldots 1 \\ j \ldots n & n-1 \ldots 0 \end{matrix}$$

   where j is the word size, at the end of the hash function, instead of the division operation, simply perform the '*logical-and*' operation on the result and $n-1$. This guarantees that no hash value will ever be greater than $n-1$.

If we have 'm' words in the lexicon, find the largest integer 'n' such that:

$$2^n + h = m$$

This method requires a hash table of size 2^{n+1}. Therefore, the hash table has an additional size of:

$$2^{n+1} - (2^n + h) = 2 \times 2^n - 2^n - h = 2^n - h$$

This will be practical if 'h' is a large number so the result: $(2^n - h)$ is somewhere between 10 to 20 percent of 'm'. For example, for m=1552, n is 10 and

$$h = 1552 - 2^{10} = 1552 - 1024 = 528$$

The hash size is 2048. Therefore, we have additional of 2048-1552=496 words, which may be okay, although it is on the high side. If m=1100, then we waste too much space, and if m=2000, the number of additional words in the hash table is 48, which is too small and is not practical.

For our lexicon with 1552 words, n = 2048, and the hash function is:

```
hash_table_size = n; /* n is a power of two. */
int simplified hashFunc (char *str)
{
        unsigned int tmp = *str++;
        do {
            tmp ^=(tmp << 3) + (int) *str;
        } while (*str++);
       return (tmp &(hash_table_size-1));
}/* hash */
```

The disadvantage of this optimization is that in the worst case, it could double the size of the hash table. If there are 2^n words in the lexicon then we require a hash table of size 2^{n+1} which is not practical for a very large lexicon.

The largest string of collision is of size 5. Details of collisions are shown in Table 6.10.

Table 6.10 Collision Report for 1552 Words with Hash Size 2049

Collision count	Number of entries	Percentage
0	1079	69.5
1	368	23.7
2	89	5.7
3	14	0.9
4	2	0.1
5	1	0.06

In Table 6.11 we have the result of the new hash function. The first entry is the time spent to search for a real word in the lexicon and the second number is the time spent to search for words in the lexicon that have an additional '1' in the initial place, guaranteed to fail.

Table 6.11 Sample Lexicon with Simplified Hash Function

No. of Iteration	Hash Table Size	CPU Time for Existing Words (Seconds)	CPU Time for Nonexisting Words (Seconds)
1	2048	0.049	0.066
10	2048	0.249	0.316
100	2048	2.249	3.366

6.10.1.4 Increasing the Number of Entries. I added the Unix word table, over 25,000 words, to the lexicon. Then the lexicon size was 26,696 words. The largest string of collisions is of size 6. Details of collisions are shown in Table 6.12.

Table 6.12 Collision Report for 26696 Words with Hash Size: 33377

Collision count	Number of entries	Percentage
0	18611	69.7
1	6138	22.9
2	1568	5.8
3	306	1.1
4	58	0.2
5	13	0.04
6	2	0.007

In Table 6.13, we have two entries per type of search: hash, binary search, and indexed binary search. The first one is the time spent for search for the real word in the lexicon and the second number is the time spent for search for words in the lexicon that have an additional '1' in initial place, guaranteed to fail.

Table 6.13 Sample Lexicon with 26696 Items in C

Iter-ations	Hash Table Size	CPU Time for Hash	CPU Time for Binary Search	CPU Time for Indexed Binary Search
1	26696	1.833/2.283	4.616/5.116	2.816/3.416
10	26696	10.182/12.682	25.482/28.165	15.432/18.699
100	26696	93.312/114.478	233.673/258.556	141.594/171.509

Finally, using all results for the hash table search will produce Table 6.14.

6.10.2 Building the Lexicon in Lisp. Common Lisp provides built-in hash facilities [Steele, 1990]. We built the entries using our lex-builder program, which generates the Lisp code to build the lexicon and the entries for the hash table. The test was the same as above. I wrote a Lisp function that takes an integer number i, as argument, and loops in the lexicon i times, and for each word s, gets its hash index h, then makes sure that the word at lexicon location h is the same as s. Here we did both tests for the original lexicon (1552 words) and later with the addition of the Unix dictionary (26,696 words).

Table 6.14 Time Report for Building a Sample Lexicon in C

Iter-ation	Lexical Entries	Hash Table Size	Total CPU Time for Success (seconds)	CPU Time per Word in (nano seconds)	Total CPU Time for Failure (seconds)	CPU Time per Word for Failure (nano seconds)
1	1552	3881	0.099	63	0.099	63
10	1552	3881	0.516	33	0.599	38
100	1552	3881	4.899	31	5.349	34
1	1552	2048	0.049	31	0.066	42
10	1552	2048	0.249	16	0.316	23.5
100	1552	2048	2.249	14.4	3.366	21.6
1	26696	33377	1.833	68	2.283	85
10	26696	33377	10.182	38	12.682	47
100	26696	33377	93.312	34	114.478	42

6.10.2.1 Lisp Report for Hash Access. For the lexicon of size 1552, and also the lexicon of size 26,696, the hash table performance is summarized in Table 6.15. For both lexicons, the report contains search times for looking up all words: one, ten, and one hundred times.

All data were obtained, both in C and Lisp, running on a Sparc 1+ machine. The Lisp program is written in *Carnegie Mellon University Common Lis* p.

Table 6.15 Time Report for Building a Sample Lexicon in Lisp

No. of Iteration	Lexical Entries	Elapsed CPU Time in Seconds	CPU Time per Word (nano seconds)
1	1552	0.14	90
10	1552	1.35	86
100	1552	13.75	73
1	26696	3.45	129
10	26696	31.26	117
100	26696	310.88	116

6.10.3 Building the Lexicon on a PC. Running Lisp, *Allegro Common Lisp for Micro Soft Windows* on a 486 PC, the data shown in Table 6.16 was generated.

Table 6.16 Building a Sample Lexicon in Lisp on a PC

No. of Iterations	Lexical Entries	CPU Time for all Words in Seconds	CPU Time per Word in Nano seconds
1	1552	0.549	353
10	1552	3.131	201
100	1552	28.781	185
1	26696	7.305	273
10	26696	68.107	255
100	26696	667.245	249

Running the C program on a PC, for 1552 lexical entries with a hash table size of 1,949, generates values given in Table 6.17. Similar to Table 6.13, we have two entries per type of search: hash, binary search and indexed binary search. The first one is the time spent for search for the real word in the lexicon and the second number is the time spent for search for words in the lexicon that have an additional '1' in initial place, guaranteed to fail.

Table 6.17 Sample Lexicon with 1552 Items in C on a PC

No. of Iterations	Elapsed CPU Time for Hash in Seconds	Elapsed CPU Time for Binary Search in Seconds	Elapsed CPU Time for Indexed Binary Search in Seconds
1	0.11/0.160	0.28/033	0.22/0.27
10	0.72/0.94	1.530/1.92	1.04/1.37
100	6.59/8.35	14.23/17.63	9.67/12.52

The average times for a word, for lexical entries of 1552, and the hash size of 1949, are summarized in Table 6.18.

Table 6.18 Sample Lexicon in C on a PC

No. of Iterations	Total CPU Time for Success	CPU Time per Word in Nano Seconds	Total CPU Time for Failure
1	0.11	70	0.160
10	0.72	46	0.94
100	6.59	42	8.35

6.11 Performance Comparison with Version Two Lexicon

This section presents the result of performance comparison of my approach and the lexicon organization in Circsim-Tutor version two [Lee, 1990].

In both systems, a Lisp function loops through all words in the lexicon and by accessing the lexicon, gets the lexical entries of the word. In version two, the lexical entries are saved in a trie, and the access function, *FINAS*, directly returns the lexical entries. In my system, the lexical information is spread amongst many tables. First, the hash table returns the index to the word table. The word table contains information about abbreviated words, phrasal information, compound information, and finally a range of indexes to the lexicon table.

Our objective was to get to the lexical information. So, after the hash table returns the word table index, we take the lexicon table index from the word table and return the content of the lexicon table.

Both tests were performed on a Sun Sparc 5 machine running *Carnegie Mellon University Common Lisp* version 17e. To get the best efficiency on both systems, all files were compiled.

Version 2: Number of words in the Circsim-Tutor version two lexicon was 1522. The real cpu time spent was 2.92 seconds. Elapsed time was 2.95 second. Average access time per word was 1918 nano second. (It took 2.92 second, or 2920000 nano seconds, to search for 1522 word: 2920000/1522 = 1918 nano second.)

In my system the number of words was 1621. The real cpu time spent was 0.05 seconds. Elapsed time was 0.07 seconds.

Average access time per word was 30 nanoseconds. It took 0.05 second, or 50000 nanoseconds to search for 1621 words: 50000/1621 = 30 nano seconds.

My method is 63 (1918/30=63) times faster than the method used in Version two. There is another **big** bonus here: much less memory is used. As Lee states in his thesis [1990]: he needs 18,955 lists to create a trie of level three. I have an overhead of only 1949 words, see Section 6.10.1.1 on page 138.

CHAPTER VII

LEXICAL ANALYZER

The lexical analyzer provides lexicon services to the parser. Analysis of the lexical properties of a word is at best ambiguous. In this chapter we discuss lexical ambiguity and how the parser deals with it. By taking advantage of the parser's ability to handle lexical ambiguity, the study provides help for spelling correction.

7.1 Lexical Analyzer

The lexical analyzer fetches the next input word, then searches the lexicon for the word to get its lexical properties such as part of speech.

In the C version of the parser, I used LEX [Lesk and Schmidt, 1978] to tokenize the input string. LEX generates programs to be used in the simple lexical analysis of the text. This program, when run, copies unrecognized portions of the input to the output, and executes the associated C action for each regular expression that is recognized. In the Lisp version of the parser, a small function does the tokenization.

Tokenization of the input string is a secondary task for the lexical analyzer. Its main task is to find the lexical entries for the word by searching the lexicon. The task gets complicated when the given word is misspelled or is part of a phrase.

Many words have multiple parts of speech. Each part of speech could have multiple senses. The lexical analyzer simply passes all the entries of the word in the lexicon to the parser. The parser, in turn, considers each one of the candidates one at a time. Any entry that violates syntactic or semantic rules is rejected.

7.2 Phrase Recognition Algorithm

The lexical analyzer must recognize phrases. A phrase consists of at least two words. Some phrases and their types:

1. Phrases used as adjectives:
    1.1. Participal Phrase: "My father was a man *known for his kindness*."
    1.2. Prepositional phrase: "The man *in the blue suit* spoke quietly", A preposition forms a prepositional phrase when it is followed by a noun: 'in the truck", "to California", "in spite of".
    1.3. Infinitive phrase: "I need shoes *to wear to the wedding.*"
2. Phrases used as adverbs: "He spoke *in the afternoon*." Note that this one is a prepositional phrase by definition (proposition + the + afternoon), but it answers the question *when* and functions as an adverb modifying the verb *spoke*.
3. Phrases used as nouns: "*Riding a motorcycle* can be dangerous", or "I like *riding a motorcycle*"
4. Verb phrase: a main verb and one or more helping verbs (auxiliary verbs), "*must have gone*...".

All words of a phrase are combined together as one lexical entry. To recognize a phrase, the lexical analyzer must combine the input words and check to see if they form a phrase.

There is a simple way of combining the input words in *two* word, *three* word, ..., *n* word phrases; where n is the number of words in the longest phrases in the lexicon, (in our lexicon n=3, as the longest phrase is: “mean arterial pressure”). Take the ith word of the input, combine it with the (i+1)st word of the input. Then, check to see if we have these two words in the hash table. If so, we have found a two word phrase. Otherwise, combine these two words with the (i+2)nd word of the input and check the hash table. If we find them in the hash table, then we have found a three word phrase. Otherwise, combine these words with the (i+3)rd. Continue the process until a phrase is found, or we have reached the (i+n)th word. In this case, we conclude that the ith word of the input is not part of a phrase. Now, start the process for the (i+1)st word of input.

Needless to say, the brute force approach, as described above, wastes a lot of time. My goal is to handle phrases efficiently. We are in an interactive environment and cannot spent too much time for recognizing phrases. To avoid unnecessary word combination and hash lookups, each individual word of a phrase has an entry in the word table with a flag indicating that it is part of a phrase. As soon as an input word that is part of a phrase is scanned, the lexical analyzer calls the phrase recognition algorithm. The phrase recognition algorithm attempts to recognize the largest phrase in the input. If a larger phrase, such as <u>in addition to</u>, contains a smaller phrase, <u>in addition</u>, both phrases are saved in the phrase list.

To ease recognition of phrases, each phrase is broken into subphrases and entered in the word table. Created subphrases entries in the word table do not have an index to the lexicon table as there are no lexical entries for them. A phrase of m words has one full entry in the word table with proper indexes to the lexicon table, see Section 6.6 on page 128. For $m>2$, the phrase is broken into m-2 subphrases of: m-1 words, m-2 words, ..., and two words. For instance, lets say we have the phrase: “on the other hand”, here m=4, therefore, we have m-2=2 (intermediate) subphrases plus the original phrase:

on the	No associated lexical entries
on the other	No associated lexical entries
on the other hand	Full lexical entries

The first two subphrases enter the word table with a flag indicating that they are part of a phrase without any index to the lexicon table. The last one has an index to the lexicon table location which will contain all its lexical entries. All individual words: *on, the, other,* and *hand* enter the word table with a flag indicating that they are part of a phrase. Since all four of them are meaningful words, their entries in the word table has index to the lexicon table. Since all entries in the word table enter the hash table, the original phrase along with all its subphrases and the individual words of the phrase can be looked up in the hash table.

Note that only meaningful words in the word table have an index to the lexicon table. In the phrase: <u>in addition to</u>, all three individual words (in, addition and to) as well as the subphrase <u>in addition</u> are meaningful words. Therefore all entries:

in

addition
in addition
in addition to

have entries in the lexicon table. In the phrase: vice versa , although we have entries:

vice
versa
vice versa

in the word table. Only the last one has index to the lexicon table.

To avoid unnecessary accesses to the lexicon, we will never attempt to look for a combination of words larger than the maximum number of words that can possibly be in a phrase.

We enter the phrase recognition process when the lexical analyzer notices that the next word of the input is part of a phrase (its phrase flag is set.) The following algorithm is used for phrase recognition.

```
Read the next word.
if the word has the phrase flag set in the lexicon, then
        words-scanned-so-far= word
        call phrase recognition process
end if
```

Phrase recognition algorithm is:

```
loop
   Read the next word.
   if end of input is reached, or a punctuation is found, or the maximum number of
   words in a phrase is exceeded then
       No phrase is recognized. Put the words in words-scanned-so-far back for later
   usage.
       exit the loop
   end if
   if the word has the phrase flag set in the lexicon, THEN
       add the word to the list of words-scanned-so-far.
       Search the hash table for words-scanned-so-far.
       If the hash table search succeeds THEN
                If the entry in the word table indicates
                that we have a full phrase THEN
                    add the words scanned so far to the phrase list.
                    We have recognized a phrase.
                    If the entry does not have the phrase flag set THEN
                    No further Phrases. Exit the loop.
                    end if
                end if
                Note if we get here, the entries in
                words-scanned-so-   far are part of a bigger phrase.
       end if
   else
       There is no phrase any more. Put the words in
       words-scanned-so-far that have not been recognized
       as a phrase back for further processing.
       exit the loop
   end if
end loop
```

Now the entries in the phase list are the recognized phrases. Currently, we use the last recognized phrase, the one with the largest number of words, as in the previous example, when we have the three words: in, addition, and to, although the above algorithm returns two phrases: in addition and in addition to, we simply pass the phrase: in addition to to the parser.

7.2.1 Example. Let's say we have the input:

in addition to the parser, the compiler has....

The lexical analyzer and the phrase recognition process do the following:

1. The first word: the 'in' entry in the lexicon indicates that it can be part of a phrase. Therefore we set the words scanned so far to 'in', and call the phrase recognition algorithm.
2. The next word: the 'addition' entry in the lexicon indicates that it could be part of a phrase. Therefore add 'addition' to the words scanned so far, giving: 'in addition'.
3. Hash lookup succeeds for 'in addition', we enter it into the recognized phrase list.
4. The entry of 'in addition' in the lexicon indicates that it could be part of a phrase. Therefore we continue the process.
5. The next word: the 'to' entry in the word table indicates that it could be part of a phrase. Therefore add 'to' to the words scanned so far, giving 'in addition to'.
6. Hash lookup succeeds for 'in addition to', we enter it into the recognized phrase list.
7. The entry of 'in addition to' in the lexicon indicates that it is not part of a larger phrase. Therefore we exit from the loop.

At the end of the loop, we have recognized two phrases: ***in addition*** and ***in addition to***.

7.3 Comment Entries

A new part of speech called *comment-entry* is added to the system. Similar to the programming language parsers, the parser simply ignores the words with part of speech *comment-entry*. Comment entries are added to the lexical analyzer so it can filter out irrelevant words that are not important. Often, we have seen the students typing: *see you later alligator*, or similar slang phrases. Comments are also used in punctuation handling.

7.4 Punctuation Handling

It is very difficult to enforce punctuation accurately except in a very obvious way. To a large extent we ignore punctuation marks, because the students use them in such an idiosyncratic way. Many users typing a sentence to be processed by a machine, do not bother to insert punctuation unless they are emphasizing a point. We take punctuation seriously in just two situations:

1. recognizing abbreviations, and

2. breaking up a phrase.

An example is provided to clarify the concept:

> They committed several serious felonies before the police caught up with them. **In addition , to** escape, they robbed a bank.

the comma between the phrase in addition and the word to causes the phrase recognition to recognize the phrase '***in addition***' which is followed by the word '***to***'

The implementation is rather simple. When the next input word is a punctuation, the lexical analyzer assigns the comment_entry as its part of speech. If we are in middle of a phrase, it simply breaks the phrase, as the next input word, the punctuation such as ',' does not exist in the phrase. If however, we have a phrase with a punctuation mark in the middle, this approach does not interfere with the phrase recognition process. The only advantage here is that the parser ignores all words that have comment_entry as their part of speech.

For abbreviations, if the word before the punctuation mark is not found in the lexicon then the algorithm looks for a period that signals a (random) abbreviation.

7.5 Handling of Spelling Errors by the Lexical Analyzer

As we mentioned in the spelling correction chapter, at first we limit our spelling error detection to isolated words. When the next input word cannot be located in the lexicon, the lexical analyzer assumes a spelling error. The spelling correction process finds those words in the lexicon that meet the minimum disagreement threshold with the misspelled word. The lexical analyzer can eliminate some of the candidates using the rule set in the spelling correction chapter (which contains rules for error weights, word boundaries, etc.) If there still remains more than one candidate, the lexical analyzer seeks the help of the parser to eliminate some of the candidate words.

In the previous sections we mentioned that the parser accepts words with multiple parts of speech. Exploiting this property of the parser, the lexical analyzer returns a list of replacement candidate words, along with their lexical entries to the parser. Since in general all the candidates do not have the same syntactic and semantic properties, the parser may be able to reject some of them. Thus the parser actively helps the spelling corrector by using the context.

CHAPTER VIII

SYSTEM IN ACTION

The previous chapters described the system components in detail. In this chapter we examine the interaction amongst the system components. Also, a few examples are presented to show the system's operation.

Section 8.1 describes the interleaved processes that perform spelling correction. Section 8.2 describes the supporting functions provided for Lexical Functional Grammar. Section 8.4 and Section 8.4.4 present a few examples of the parser operations.

8.1 Cooperative Processes for Spelling Correction

The interleaved processes:

- lexical analysis
- spelling correction
- syntax analysis
- semantic analysis

work together to find the best matching word from the lexicon for a misspelled word.

The steps for spelling correction and the choice of the best candidates are organized as follows:

1. The lexical analyzer detects that the next input word is misspelled. It passes the word to the spelling corrector.
2. The spelling corrector returns a list of one or more replacement words for the misspelled word. The list is sorted in ascending order of calculated error weights. We refer to this list as the *spell_list*. The *spell_list* is simply a list of words:

   $$(w_1\ w_2\ \ldots\ w_n)$$
3. The lexical analyzer passes the list to the phrase recognition to check whether any word in the list can be combined with the previous/next input word(s) to form a phrase. If a phrase can be constructed, the word that is used in the phrase is considered the only replacement candidate for the erroneous word and is passed to the parser. Otherwise,
4. The lexical analyzer sorts the words in the *spell_list* by their parts of speech (pos). For each *pos*, there is a list of replacement words from *spell_list*:

   $$(pos_1\ \ w_i\ \ldots\ w_j)$$

   We refer to this list as the *pos_list*. The words in this list remain sorted in ascending order of their error weight. Then, the lexical analyzer creates a list of all *pos_lists*, named *lex_list*. The sub-lists *pos_lists* are put in lex_list in ascending order of the first word in the sub-list:

   $$((pos_1\ \ w_i\ \ w_j\ \ w_k)\ (pos_2\ \ w_l\ \ w_m\ \ w_n)\ \ldots\ )$$

   In this list, the error weight of w_i is smaller than the error weight of w_j and the error weight of w_i is smaller than the error weight of w_l. There is no enforced relation between the error weight of w_j and the error weight of w_l, or w_m. It could be bigger or smaller. They could also be the same.

5. The parser examines each sub-list: *pos_list* of *lex_spell*, and discards the sub-lists that violate the syntactic rules. Obviously, the syntax analysis process cannot distinguish among the words with the same part of speech. Thus, the parser either accepts or rejects a *pos_list*.
6. The parser shortcomings are overcome by the Lexical Functional Grammar rules. The LFG f-structure construction process uses the LFG annotations in the lexicon and the grammar to examine the validity of the candidates with the same part of speech. Specifically, enforcement of the three well-formedness conditions (see Section 2.2.6 on page 11 for details) is helpful here.

Now we describe the cooperation amongst the processes (lexical analyzer, phrase recognition process, syntax analyzer and semantic analyzer) for spelling correction in reference to examples covering different aspects of the method.

8.1.1 Use of the Phrase Recognition. Suppose the user types a sentence containing the following two words:

"card contr"

The lexical analyzer concludes that 'card' is a misspelled word, since we do not have the word 'card' in our lexicon. It calls the spelling error process to find a replacement for the word 'card'. The spelling corrector returns a list of words shown in Table 8.1 for the misspelled word 'card'.

Table 8.1 Spelling Correction Candidates for 'card'

Weight	Matching Word
36	**cardiac**
61	careful
61	carried
94	carbon
95	case
95	said
95	vary
105	are
106	carotid
107	cause
110	acts
117	arch
119	caused

The lexical analyzer passes the list to the phrase recognition process. The first word in Table 8.1 could be part of a phrase. The phrase recognition, in turn, calls the lexical analyzer to fetch the next input word. The lexical analyzer, fetches the word 'contr', and because it cannot locate it in the lexicon, calls the spelling corrector. The spelling corrector returns a list of words shown in Table 8.2.

Table 8.2 Spelling Correction Candidates for 'contr'

Weight	Matching Word
24	control
36	contract
36	controls
48	contracts
60	controlled
60	controller
69	content
72	contractile
72	contracting
72	contraction
94	central
94	contain
96	**contractility**

The lexical analyzer passes all the replacement words to the phrase recognition process. The phrase recognition, in turn, concludes that we have the phrase:

"cardiac contractility"

Note that the word "*cardiac*" is the first choice for the misspelled (actually abbreviated) word 'card' (lightest error weight), and the word "*contractility*" is the last choice for the misspelled (actually abbreviated) word 'contr'. However, since they make up a phrase, the lexical analyzer simply returns a single choice *"cardiac contractility"* to the parser.

8.1.2 Use of the Parser. If the user types the following sentence:

"lohng sees Mary"

The lexical analyzer calls the spelling error to correct the misspelled word: *lohng*. The spelling corrector returns a list of words shown in Table 8.3.

The lexical analyzer passes the above list to the parser. The parser, in turn, only accepts words with part of speech 'noun', as the other parts of speech are not

Table 8.3 Spelling Correction Candidates for ‘lohng’

Weight	Matching Word	Part of Speech
80	long	adj, adv , noun, verb
104	longer	adj
129	linger	verb
150	John	noun
174	length	noun

syntactically correct. Therefore the list is reduced to three words as shown in Table 8.4.

Note that the parser cannot distinguish amongst the three words, in Table 8.4, as they all are of type ‘noun’.

Table 8.4 Candidates for “lohng” Reduced by the Parser

Weight	Matching Word	Part of Speech
80	long	noun
150	John	noun
174	length	noun

8.1.3 Use of the LFG Information. The parser passes the list in Table 8.4 to the LFG f-structure construction process. The word *sees* requires a *subject* . The word *‘John* ’ is the only word that could be subject for the word *sees*. Therefore, the f-structure builder process eliminates the words ‘*long*’ and ‘*length* ’ from consideration. Thus the system concludes that the only replacement word for ‘lohng’ is ‘John’.

8.2 LFG Supporting Functions

Many functions are defined that can be inserted in the grammar for LFG support. Here is a brief description of them:

lfg_plant_equation (equation) Plants the functional equation: *equation* into the f-structure of the adjacent node in the grammar. The *equation* argument must be a valid, and acceptable lfg equation of the form (attribute value). If this function is augmented before node ’X’ in a grammar rule, it inserts the equation in X’s f-structure, as follows:

$$(\downarrow\text{attrib}) = \text{value}$$

lfg_plant_constrain_equation(attrib value) Plants constraining equation (attribute value) in its adjacent node. This is similar to the previous function: *lfg_plant_equation* () but the equation is a constraining equation: $(\downarrow\text{attrib}) =_c \text{value}$

lfg_equate_up_with_down_arrow () This is $\uparrow = \downarrow$. We are to equate the f-structure of the adjacent nonterminal and the f-structure of the mother node (left hand side nonterminal node.)

lfg_equate_field_with_down_arrow (attribute) This function performs the operation: $\uparrow$attribute = $\downarrow$

lfg_extract_attName_equate_field (attribute) This is an extension to the LFG to allow context dependent items to be loaded into the f-structure. This function performs as:

$$\uparrow(\downarrow\text{attribute}) = \downarrow$$

For example ($\uparrow(\downarrow SUBJ$))=$\downarrow$ gets the corresponding value of attribute 'SUBJ' in the f-structure of the node, call it VAL. Now plant the equation $(\uparrow VAL) = \downarrow$. *VAL* must be a simple symbol, as it will be the attribute of the new functional entry: $(\uparrow \text{VAL}) = \downarrow$. An example at the end of this section clarifies the usage of this function.

lfg_equate_upAtt_downAtt(upAtt downAtt) Performs the following function: $(\uparrow\text{upAtt}) = (\downarrow \text{downAtt})$

lfg_PRED_plus_equate_UP_down (insert_equation) This is a convenient function that first plant the equation: *insert_equation* in $\downarrow$ and then performs the operation: $\uparrow = \downarrow$.

lfg_plantConstrain_equate_filed (attrib1 value attrib2) A double action: First, plants constraining equation:

$$(\downarrow\text{attrib1}) =_c \text{value}$$

Then it performs the following:

$$(\downarrow\text{attrib2}) = \text{value}$$

Since the operation $\uparrow = \downarrow$ is frequently used, I have defined the action: *!LFGequateUpDown* which performs the operation. This is for user convenience.

As mentioned in Chapter IV, the parser requires that all inserted functions in the grammar return a boolean result (success or failure). As the input words are parsed, a parse tree is terminated as soon as a syntax rule is violated, or any one of the inserted functions fails.

The function: *lfg_extract_attName_equate_field* is the most complicated LFG function in the system. To clarify its usage an example is provided to show the usage of this function. Let's say we have the following grammar rule:

x $\longrightarrow$ {{*lfg_extract_attName_equate_field* ('att)}} y

If f_m is y's f-structure, and f_n is x's f-structure, before we apply the function:

$$f_m \begin{bmatrix} A_1 & V_1 \\ att & w \\ A_n & V_n \end{bmatrix} \qquad\qquad f_n \begin{bmatrix} B_1 & C_1 \\ & \\ B_i & C_j \end{bmatrix}$$

then applying the function modifies f_n to:

$$f_n \begin{bmatrix} B_1 & C_1 \\ w & \begin{bmatrix} A_1 & V_1 \\ att & w \\ A_n & V_n \end{bmatrix} \\ B_i & C_j \end{bmatrix}$$

8.2.1 Sample Usage. Figure 8.1 shows a small fragment of English grammar. The actual input file for the parser generator ALI, for this grammar is shown in Figure 8.2.

$$\begin{array}{llcc}
1. & S \rightarrow & \underset{(\uparrow SUBJ) = \downarrow}{NP} & \underset{\uparrow = \downarrow}{VP} \\
2. & VP \rightarrow & \underset{\uparrow = \downarrow}{V} & \left(\underset{(\uparrow OBJ) = \downarrow}{NP}\right) \\
3. & NP \rightarrow & \left(\underset{\uparrow = \downarrow}{DET}\right) & \underset{\uparrow = \downarrow}{N}
\end{array}$$

Figure 8.1 Small Grammar
Adapted from Wescoat [1987]

8.2.2 Default LFG functions To allow incremental addition of words to the lexicon, the system allows words to enter the lexicon with or without part of speech. Further, the semantic structure (lexical function) for the word can be left blank. The system inserts default lexical functions at run time.

The difficulty here is that there may be many words that have no lexical function entries in the lexicon. When a sentence contains two or more words without

```
/*
** Michael Wescoat's grammar
**/
#include <stdio.h> /* Standard I/O includes */
#include "gram.h" /* grammar include file */
%%
 !! No keyword definition.
%%
 !! No Special keyword definition.
%%
S ::= {{ (lfg_equate_field_with_down_arrow 'SUBJ) }}
          NP
          !LFGequateUpDown     VP

NP ::= !LFGequateUpDown     det
         !LFGequateUpDown     noun

NP ::= !LFGequateUpDown      noun

VP ::= !LFGequateUpDown     verb

VP ::= !LFGequateUpDown     verb
          {{ (lfg_equate_field_with_down_arrow 'OBJ ) }}       NP
%%

%%
;; /* No functions provided. */
```

Figure 8.2 Small Grammar Represented for ALI

lexical functions, the system should be able to continue. This requires that two or more lexical functions be unified. With this in mind, I created a Lisp function to generate a lexical function for a given word *x* as follows:

(DEFAULT-LEX_FUNC ((x EMPTY)))

This way if two words *x* and *y* (with no lexical functions) have to be unified: we get the following lfg f-structure:

(DEFAULT-LEX_FUNC ((x EMPTY) (y EMPTY)))

8.3 Grammars Used

Several different grammars have been used to test my parser. Tomita's grammar [1987] was used for the sample example in Section 3.4 on page 45 and also for more serious problem of ambiguity in Section 3.5 on page 56. The lexicon needed for this grammar was nine words as shown in Table 3.2.

Michael Wescoat's [1987] sample grammar, shown in Figure 8.1, was used along with the lexicon shown in Figure 8.3.

This grammar and the lexicon were used to test preliminary LFG f-structure generation. Also, this grammar was used in testing my ideas about cooperative processes in spelling correction.

John N ($\uparrow$PRED)= 'JOHN'
($\uparrow$NUM) = SING
($\uparrow$PERS) = 3

sees V ($\uparrow$PRED)='SEE<($\uparrow$SUBJ) ($\uparrow$OBJ)> '
($\uparrow$SUBJ NUM) = SING
($\uparrow$SUBJ PERS) = 3

Mary N ($\uparrow$PRED)= 'MARY'
($\uparrow$NUM) = SING
($\uparrow$PERS) = 3

Figure 8.3 Small Lexicon

The Circsim-Tutor Version Two grammar was used for some of the transcripts of the system, see Seu [1992]. The Version Two grammar is:

S $\rightarrow$ adv NP VP | NP VP
| comp S | aux NP VP
| NP VP conj S
| **S S**
NP $\rightarrow$ noun NP | det noun
| noun | ADJS noun
| det ADJS noun
| det ADJS noun PP
| pron
| noun conj noun | NP PP
VP $\rightarrow$ verb prep | verb pron
| av verb NP
| av verb PP | verb PP
| av verb | verb NP PP
| verb S | verb adv
| av adv verb | verb | av verb
| verb NP
PP $\rightarrow$ prep NP [PP]
ADJS $\rightarrow$ adj ADJS | adj
ADVP $\rightarrow$ adv | adv adv

Note that the parser can handle the production: **S $\rightarrow$S S** .

A more current grammar was designed by Michael Glass [1994] for the Circsim-Tutor version III. This grammar is called the 'Answer Grammar' and it is used to parse student answers to system questions. Our final lexicon is around 4000 words. The grammar is not finalized yet. A draft of the current grammar is:

S $\rightarrow$ adv NP VP | NP VP
| comp S
| aux NP VP | NP VP conj S
| sent
ADJS $\rightarrow$ adj ADJS | adj
N $\rightarrow$ noun N | noun | pron
NP $\rightarrow$ N | det N | ADJS N
| det ADJS N
| N PP | det N PP

```
               | ADJS N PP
               | det ADJS N PP   | NP conj NP
               | NP PP
P          →  prep NP
SBAR       →  comp S
VP         →  verb prep   | V1 NP
               | V1 PP PP  | V1 PP
               | V1 NP PP  | V1 NP NP
               | V1 SBAR
               | VP adv  | adv VP
               | V1
V1         →  modal [not] V2   | V2
V2         →  perf [not] V3-en    → V3
V3-en      →  vbeen [not] V4-ing    | V4-en
V3         →  vbe [not] V4-ing   | V4
V4-ing     →  vbeing [not] V-en   | V-ing
V4         →  vbe [not] V-en   | V
V4-en      →  vbeen [not] V-en   | V-en
V-en       →  ven | vbeen
V          →  verb | vbe
V-ing      →  ving | vbeing
```

In addition to the usual terminal symbols, a few additional parts of speech are added to the grammar to facilitate the coverage of the verbs. Table 8.5 lists the new parts of speech added to the grammar.

Table 8.5 New Parts of Speech in Circsim-Tutor Grammar

Part of Speech	Sample Entry
modal	should, might
not	not
perf	have, had
sent	no change
vbe	be, is, am, are, was, were
vbeen	been
vbeing	being
ven	increased, added,
verb	think, seem, sees, kills, falls
ving	increasing, adding

The above examples show the benefit of having a table driven system. Changing the grammar or the lexicon is just a matter of running a makefile. A few minutes later the system is ready to parse the sentences in the new grammar.

I designed the parser and the lexicon aiming at a very large grammar and a large lexicon. To test the behavior of the system for a large lexicon, I used the Unix Dictionary (the largest number of words that I could put my hand on). As reported in Section 6.10 on page 136, the system operated very fast and showed the promise of much higher capacity.

The parser does not have any limit on the grammar size. It can efficiently handle a grammar with thousands of rules and nonterminal symbols.

To test the behavior of the parser in a programming environment, I choose the VHDL [Baker , 1993] grammar with 750 productions, 250 nonterminals, and 150 terminals. I ran the parser on a few VHDL programs. The parser worked efficiently.

8.3.1 Handling Ellipsis. In an attempt to see the parser behavior for a very large grammar, I used the following grammar (presented in Fischer and LeBlanc [1988, p. 213]):

$$S \rightarrow X_i \, Z_i \qquad 1 \leq i \leq n$$

$$X_i \rightarrow Y_j \, X_i \mid Y_j \qquad 1 \leq i, j \leq n, \; i <> j$$

As we saw in Table 2.2, the grammar generates a huge parse tables for n=10 in LR parsers. I loaded the above grammar for n=100.

To use this grammar, without changing the lexicon (not adding Y_j and Z_i) and test the parser for ellipsis, I added a dummy production:

$$dummy\_unreachable \rightarrow verb$$

With the added production, we have the following:

- Terminals: 217 symbols.
- Nonterminals: 102.
- Number of productions: 10,102.

I parsed the word: “saw”. This is obviously a fragment. Therefore, all nonterminals are considered to find a match. It took 0.36 seconds to generate:

(dummy_unreachable (verb ‘saw’))

Note that to recognize a sentence fragment, all productions of the grammar must be considered. This test shows not only that the parser can handle a very large grammar, over 10,000 rules, but also it can competently handle fragments.

8.4 Parser in Action

A few sample sentences are used here to show the parser in action.

8.4.1 Some Extracts from Circsim-Tutor. We show the result of the parser operating on actual student input. For simplicity, only the parse trees are shown.

8.4.2 An Extract from Circsim-Tutor. In one of the sessions, the following conversation took place between the Student and the Tutor. We show the result of the parser operating on actual student input. For simplicity, only the parse trees are shown.

Tu: What are the determinants of mean arterial pressure?
St: total peripheral resistance and cardiac output
Tu: Correct, mean arterial pressure is determined by total peripheral resistance and cardiac output.
Tu: What is the important determinant in this case?

St: total peripheral resistance
Tu: Correct, the important determinant of mean arterial pressure in this case is total peripheral resistance.
Tu: What is the relationship between total peripheral resistance and mean arterial pressure?
St: mean arterial pressure depends on total peripheral resistance

Parsing: *total peripheral resistance and cardiac output* .

This is a fragment. It is recognized as an NP.

```
(NP  (noun 'total peripheral resistance')
     (conj 'and')
     (noun 'cardiac output'))
```

Parsing: *total peripheral resistance* :

This is a fragment. It is recognized as an NP:

```
(NP (noun 'total peripheral resistance')
```

Parsing: *mean arterial pressure depends on total peripheral resistance* .

```
(S  (NP (noun 'mean arterial pressure'))
    (VP (verb 'depends')
        (PP (prep 'on')
        (NP (noun 'total peripheral resistance')))))
```

8.4.3 A Simple Example of LFG. To show the way the system uses the LFG f-structure construction, we present one example using a simple grammar from Figure 8.1, and the lexicon from Figure 8.3. The grammar and the lexicon are from Wescoat [1987].

Parsing: *john sees Mary*

This generates the c-structure shown in Figure 8.4, and generates the following f-structure:

```
((PRED "see" SUBJ OBJ)
 (SUBJ ((PRED "John")
        (NUM SING)
        (PERS 3)))
 (OBJ ((PRED "Mary")
       (NUM SING)
       (PERS 3))))
```

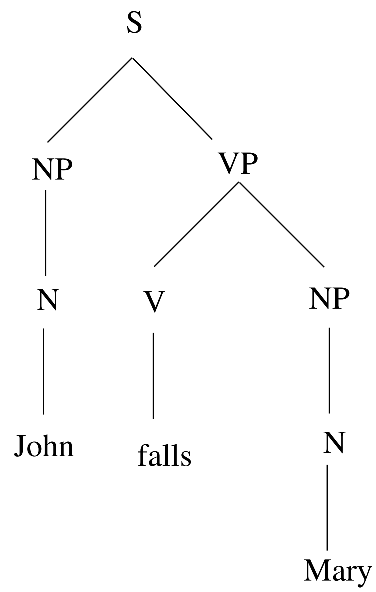


Figure 8.4 LFG C-structure for "John Falls Mary"

8.4.4 More Complex Examples. This section covers examples of sentences that contain words with multiple senses.

Parsing the sentence:

blood was added to the atrium by the venous return

Generates the c-structure shown in Figure 8.5 and the following f-structure:

```
((PRED add BY-PP SUBJ TO-PP)
  (SUBJ ((PRED blood)))
  (TO-PP ((PCASE TO-PP)
          (PRED atrium))))
(BY-PP ((PCASE BY-PP)
          (PRED vr))
  (PASSIVE +))
```

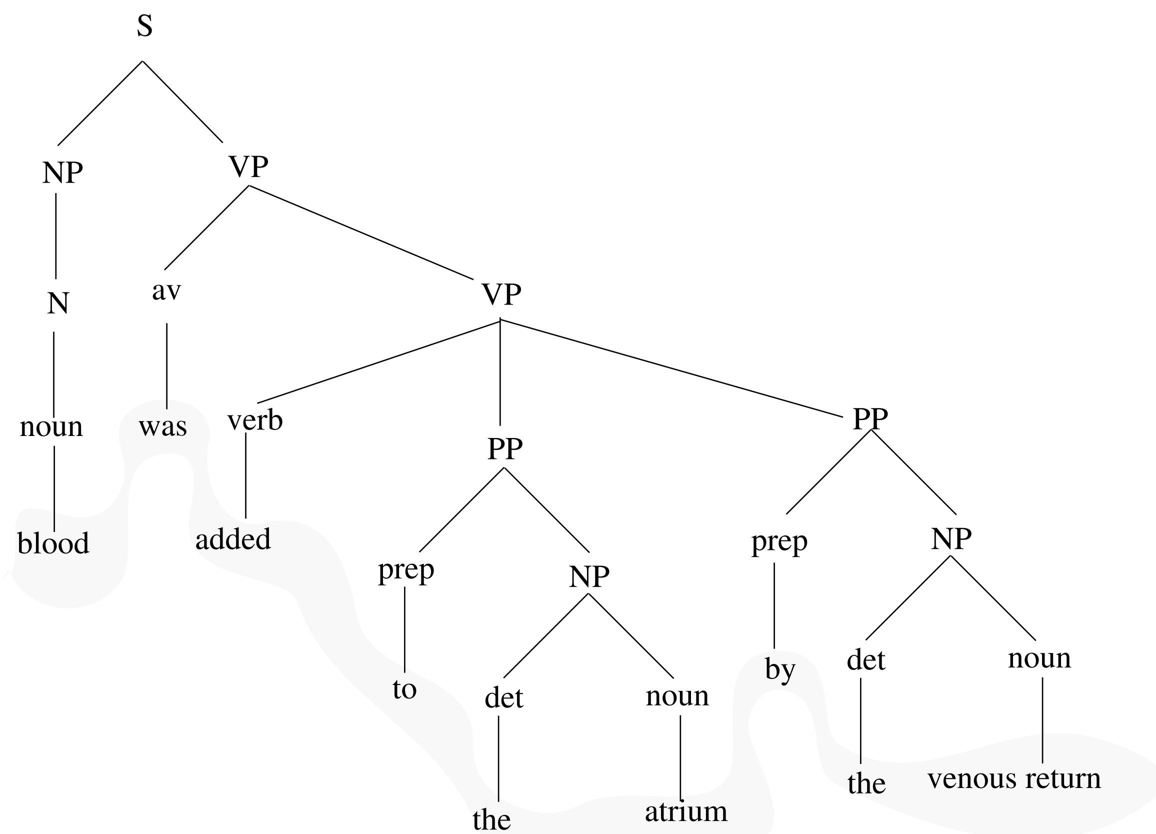


Figure 8.5 C Structure for the Sample Sentence:
"blood was added to the atrium by the venous return"

Parsing the sentence:

I think that blood was added to the atrium by the venous return.

generates the c-structure shown in Figure 8.6, the system generates the following f-structure:

```
((PRED think SUBJ COMP-THAT)
 (SUBJ ((HUMAN +)))
  (COMP-THAT ((PRED add BY-PP SUBJ TO-PP)
                (PASSIVE +)
               (SUBJ ((PRED blood)))
               (TO-PP ((PCASE TO-PP)
                       (PRED atrium))))
             (BY-PP ((PCASE BY-PP)
                       (PRED vr)))
             (BY-PP ((PCASE BY-PP)
                       (PRED vr)))
```

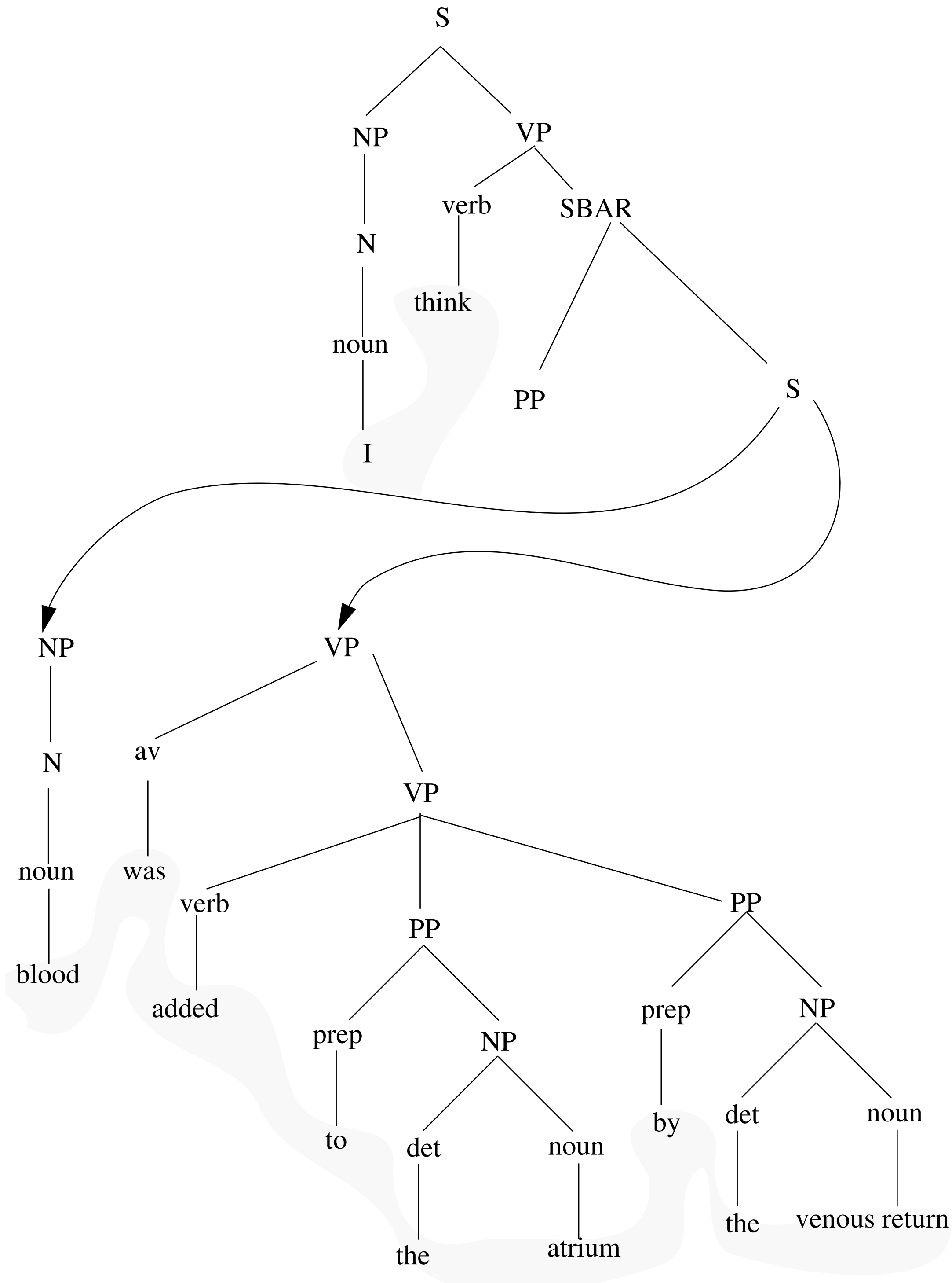


Figure 8.6 C Structure for the Sample Sentence:
I think that blood was added to the atrium by the venous return.

Parsing the sentence:
the increase in reflex response adds blood to the atrium.

generates the following f-structure:

```
((PRED add SUBJ OBJ TO-PP)
 (SUBJ ((PRED increase3)
        (IN-PP ((PCASE IN-PP)
                (PRED rr)
                (STAGE +)
                (TIME +)))
       (NUM SING)))
 (OBJ ((PRED blood)))
 (TO-PP ((PCASE TO-PP)
         (PRED atrium))))
```

and the c-structure:

```
(S (NP (NP (the:(det))
           (increase:(noun)))
       (PP (in :prep)
           (NP (reflex response :noun) ) ) )
   (VP (adds :verb)
       (NP (blood :noun) )
       (PP (to :prep)
           (NP (atrium :noun)))))
```

Note that using *increase* as a noun can require a parameter as in the sentence:
the increase in vr adds blood to the atruim.

which enerates the following f-structure:

```
((PRED add SUBJ OBJ TO-PP)
 (SUBJ ((PRED increase4)
        (IN-PP ((PCASE IN-PP)
                (PRED vr)
                (PAR +)  ))
       (NUM SING)))
 (OBJ ((PRED blood)))
 (TO-PP ((PCASE TO-PP)
         (PRED atrium))))
```

and the c-structure:

```
(S  (NP  (NP (the:(det))
             (increase:(noun))))
         (PP (in :prep)
             (NP (vr :(noun) : venous return )))
    (VP (adds :verb)
        (NP (blood :noun) )
        (PP (to :prep)
            (NP (atrium :noun)))))
```

Parsing the sentence:
blood was added to the atrium by the increase in the venous return.

generates the following f-structure:

ollowing f-structure:

```
((PRED add BY-PP SUBJ TO-PP)
 (BY-PP ((PRED increase4 IN-PP)
             (IN-PP ((PAR +)
                        (PCASE IN-PP)
                        (PRED vr)))))
 (SUBJ ((PRED blood)))
 (TO-PP ((PCASE TO-PP)
             (PRED atrium)))
(PASSIVE +))
```

and the c-structure is:

```
(S (NP (blood:(noun)))
    (VP (was:(av))
         (VP (added:(verb))
              (PP (to:(prep))
                   (NP (the:(det)) (atrium:(noun))))
             (PP (by:(prep))
                   (NP (NP (the:(det))
                              (increase:(noun)))
                   (PP (in:(prep))
                        (NP (the:(det))
                             (venous return:(noun)))))))))
```

Parsing the sentence:

the increase in venous return in reflex response adds blood to the atrium.

generates the following f-structure:

```
((PRED add SUBJ OBJ TO-PP)
 (SUBJ ((PRED increase4 IN-PP)
          (IN-PP ((PRED vr)
                    (PCASE IN-PP)
                    (PAR +)))
          (STAGE-PP ((PCASE STAGE-PP)
                         (PRED rr)
                         (STAGE +)
                         (TIME +)))
          (NUM SING)))
 (OBJ ((PRED blood)))
         (TO-PP ((PRED atrium)
 (PCASE TO-PP)))
 (TENSE PRES))
```

and the c-structure:

```
(S (NP (NP (NP (the:(det))
                    (increase:(noun)))
              (PP (in:(prep))
                    (NP (vr:(noun)))))
         (PP (in:(prep))
              (NP (rr:(noun)))))
    (VP (adds:(verb))
         (NP (blood:(noun)))
         (PP (to:(prep))
```

```
                    (NP (the:(det))
                        (atrium: (noun))))))
```

8.4.4.1 Using the Answer Grammar. Using the Answer Grammar, lets parse the sentence:

The increase in vr should have been adding blood to the atrium.

Generates the follwing f-structure:

```
((PRED add SUBJ OBJ TO-PP)
 (SUBJ ((PRED increase4 IN-PP)
        (IN-PP ((PRED vr)
         (PCASE IN-PP)
         (PAR +)))
         (DEFAULT-LEX_FUNC ((the EMPTY)))
         (NUM SING)))
 (OBJ ((PRED blood)))
 (TO-PP ((PRED atrium)
         (PCASE TO-PP)
         (DEFAULT-LEX_FUNC ((the EMPTY)))))
 (DEFAULT-LEX_FUNC ((should EMPTY)
                    (have EMPTY)
                    (been EMPTY)))
 (TENSE PRES))
```

Note that there are no LFG annotations in our lexicon for the words:

the, should, have, and *been*

and the system has inserted the default lexical function:

(DEFAULT-LEX_FUNC ((“word string” EMPTY)))

for every word that has no lexical function.

For the above sentence, the generated C-structure, using the answer grammar is:

```
(S (NP (NP (The:(det) Matching: the)
           (N (increase:(noun))))
 (PP (in:(prep))
     (NP (N (vr:(noun) Matching: venous return)))))
 (VP (V1 (should:(modal))
         (V2 (have:(perf))
             (V3-en (been:(vbeen))
                    (V4-ing (V-ing (adding:(ving)))))))
      (NP (N (blood:(noun))))
      (PP (to:(prep))
          (NP (the:(det)) (N (atrium:(noun)))))))
```

CHAPTER IX

CONCLUSIONS AND DIRECTIONS FOR FUTURE RESEARCH

9.1 Significance of the Research

This research focuses on the construction of a practical natural language understanding system with minimal limitations. As a result, a parser was created that meets the challenges facing a natural language parser in a dialogue with a user who is typing input on a keyboard. Like any casual user of a computer our medical students make errors in typing, spelling, grammar, and diction all the time. The parser handles all sorts of ambiguity, and maintains multiple parses without parsing any part of the input twice.

The thesis introduces a spelling correction algorithm that uses the sentence context for error recovery. The algorithm interleaves parser actions with the process of spelling error correction.

There is no restriction on the grammar. There is practically no limit on the grammar size nor any restrictive rule for ambiguous productions. There is no limit on the number of terminal/nonterminals or the number of productions. As we saw in Chapter II, formula (1) on page 15, if p is the number of productions in a grammar and t is the number of terminal symbols in the grammar, and assuming there are 8 bits per byte, then, the table size, in bytes, will be:

$$LLsize = \frac{p \times t}{8}$$

The lexicon organization was designed with a very large lexicon in mind. There is no limit on the number of senses that a word can have. The most important design criterion was to provide efficient access time.

The next sections present a summary of my achievements in parsing, spelling correction, and lexical analysis.

9.1.1 Parser. The main strength of this parser is its ability to handle all natural language parsing requirements, as specified in the introduction. Namely, it:

- demonstrates speed and space efficiency.
- allows augmentation of semantic rules in the grammar. The parser enforces the semantic rules. Also, as soon as any semantic rule is violated in a branch, the parsing of the branch can be terminated.
- handles all sorts of ill-formed input such as fragments and ellipses.
- recognizes the phrases within a sentence.
- accepts input with multiple parts of speech.
- uses syntactic/semantic information to help correct the spelling errors.
- deals with all sorts of ambiguous sentences, as natural language is naturally ambiguous.

This parser has proved to be both robust and efficient. It can handle the errorfilled input that medical students throw at it without keeping them waiting. The parser uses a minimal amount of storage. It has also preserved the LL time efficiency and the rich LL structure to be used later for error recovery.

The parser can handle ambiguous grammars and all sorts of fragmentary input. The parser is tightly coupled with the lexical analyzer. This approach reduces the lexical ambiguity and allows us to make the parser aid the spelling correction.

A new parsing algorithm was created that combines a top down parsing algorithm with chart parser techniques. The natural LL(1) limitation of left recursive handling is overcome by bottom up construction of the parse tree when we hit a left recursive production. The augmentation of Lisp functions in the grammar, and their evaluation as the parser progresses, allows generation of Lexicon Functional Grammar f-structure.

To illustrate the concept, a simple grammar rule and some lexical entries are provided here.

$$\begin{array}{ccc} S \rightarrow & NP & VP \\ & (\uparrow \text{SUBJ}) = \downarrow & \uparrow = \downarrow \end{array}$$

Figure 9.1 shows a sample of lexical entry for the transitive sense of the verb “increases”.

Representation of item

Syntactic category

increases V ($\uparrow$PRED)= ‘increase<($\uparrow$SUBJ) ($\uparrow$OBJ)> ’
($\uparrow$SUBJ NUM) = SING
($\uparrow$SUBJ PERS) = 3
($\uparrow$OBJ PAR) = $_c$ +

List of functional schemata

Figure 9.1 Format of LFG Lexical Items

After the parse tree is built, all LFG functional schemata, such as the one shown above for the verb ‘increases’, are copied to the parse tree’s leaf nodes. Now two Lisp functions inserted in the rule, propagate the information from the leaf nodes to the top node *S*. The first function performs the operation: $\uparrow$SUBJ)=$\downarrow$, and the second function performs the operation: $\uparrow$=$\downarrow$. The employment of LFG concepts, most important, the three well-formedness conditions on f-structure: *uniqueness, completeness*, and *coherence* conditions, has broadened the parser capability to deal with ill-formed and ambiguous input.

The parser allows each lexical entry to have multiple parts of speech. Since a word can have several *senses* for a single part of speech, the system handles separate semantic structures for each *sense* of a given word.

It is clear that the context free approach is insufficient to deal with a natural language such as English because different words have different requirements. For instance, a transitive verb requires a subject and an object, whereas an intransitive verb requires only a subject. Lexical Functional Grammar has enabled the context free parser to deal with context sensitive English sentences. This is done by having grammatical requirements for each word of the lexicon. For instance, the word *increase* has multiple parts of speech: 1) *noun*, 2) *verb*. There are two senses for *increase* as a *verb*:

1. Intransitive form for *increase*. In this case there should not be an object in the sentence. Further, the subject must be a parameter. For example, *Heart rate increases.*
2. Transitive form for *increase*. In this case there should be an object in the sentence. Further, the object must be a parameter. For example, *The baroreceptor reflex increases heart rate*.

It is not possible to enforce the above rules in the parser. We cannot have specific rules for each sense of a word because we need to deal with all sorts of word combinations. This will simply lead to an unacceptably large number of possibilities.

Using Lexical Functional Grammar, on the other hand, we can save all of the above requirements in the lexical entries for the verb *increase*. Figure 9.1 shows the lexical functions for the transitive sense of increase. Figure 9.2 shows the lexical functions for the intransitive sense of increase.

increases V (↑PRED)= 'increase<(↑SUBJ)>'
(↑SUBJ NUM) = SING
(↑SUBJ PERS) = 3
(↑SUBJ PAR) = $_c$ +

Figure 9.2 Lexical Function for Intransitive Sense of Increase

Once the parser builds the parse tree of a sentence in which *increase* is the main verb, the following actions take place:

1. If there is no object in the sentence, use the intransitive *sense* of *increase*. Further, we check the lexical function of the *noun* that is the subject of the sentence. If it is not marked 'PAR +', it is not a parameter, so we simply reject the sentence. If however, the subject is of type PAR, the sentence is considered valid. For example, the system will reject the input:

   The baroreceptor reflex increases.
2. If there is an object in the sentence, use the transitive *sense* of *increase*. Further, we check the lexical function of the *noun* that is the object of the sentence. If it is not marked 'PAR +', it is not a parameter, so we simply reject the sentence. If however, the object is of type PAR, the sentence is considered valid. For example, the system will reject the input:

   The heart rate increases the baroreceptor reflex.

The subject of the sentence for both senses of *increase* must be a third person singular. Another benefit is to match the parts of the sentence with the verb arguments.

9.1.2 Spelling Correction. The spelling correction algorithm provides practical spelling correction that is both fast and reliable. We have improved spelling correction in three ways:

1. Combining parsing and spelling correction, which allows the parser to aid the spelling correction. This step uses syntactic information, the phrase recognition process, and semantic rules to improve the spelling correction output. Most important, unlike the first version of Circsim-Tutor, the spelling corrector returns a list of replacement words instead of a single replacement word. The parser then picks the best choice.
2. Assignment of error weights for string comparisons that take keyboard effects into consideration. The choice of position sensitive error weights facilitates recognition of word boundary problems as well as random abbreviations. When there is more than one replacement candidate word, having an error weight for each one of them enables the spelling corrector to rank the replacement candidates for the misspelled word.
3. Extending the three way match algorithm to the four way match algorithm to achieve a more realistic comparison of two words and produce a better correction of the first word to transform it to the second word. Note that the four way match uses only two character look ahead as in the original algorithm, so it does not add complexity.
4. Segmenting the lexicon into 26 partitions. In Chapter II, we saw that the statistics support the hypothesis that most errors happen at positions other than the initial characters. To exploit this characteristic of spelling errors the lexicon is treated as a circular array of 26 segments. We start the search in the segment with the same first character as the misspelled word.

The spelling correction method can handle all sorts of errors:

1. Typographic errors: In this type of error, the user may well know the correct spelling and the error happens when the user hits the wrong key, for example ‘teh’ for ‘the’ or ‘boqk’ for ‘book’.
2. Phonetic errors: In this case, the user does not know the correct spelling, but substitutes a phonetically correct string, ‘physikal’ for physical, or worse ‘fizykal’, or ‘nacheral’ for ‘natural.’ This type of error occurs when a person does not know the exact spelling of a word, and resorts to phonetic spelling for the word.
3. Cognitive errors: this type of error is based on a misconception or a lack of knowledge. Examples are: ‘recieve’ for ‘receive.’

Special attention is given to word boundary problems as well random abbreviation.

The spelling correction routine has improved the method of calculating the error distance between two words. For every mismatch an error weight has been assigned. The error weights are assigned with respect to the likelihood of the errors; the easier an error is to make, (e.g., keyboard effects) the lighter the error weight will be.

Many exciting results were brought to the surface by the spelling correction research. To name a few, we were able to partition the lexicon to lower the number of comparisons for spelling correction. Keyboard information as well as phonetic information is used for character comparison.

If the spelling corrector determines that there is more than one acceptable candidate, then the parser takes all candidates and tries to reduce the number to one, rejecting the ones that violate syntactic or semantic rules. If this step does not reduce the number to one, then the first remaining candidate word is chosen. This allows us to avoid asking users to intervene most of the time. When the misspelt word happens to be in a phrase, the phrase recognition process can also help narrow the choices. A few examples are provided to clarify the concept.

1. For the erroneous word 'ater', two candidates 'after' and 'ate' have the same error distance (one needs insertion of 'f', and the other deletion of 'r'). The parser could be used to select 'ate' in the sentence:

   ate

   She ~~***ater***~~ *the Cake.*

2. The phrase recognition could be used to correct the two misspelled words: "*cardi contr*" to

   cardiac contractility.

3. In the sentence:

   lohng sees Mary

   the replacement words for 'lohg' could be: "long, longer, linger, John, length". The parser eliminates words: "longer, linger", since we expect a noun for the replacement. Then the LFG f-structure builder will eliminate the remaining words (*long, length*) since they cannot be the subject of *sees*. Cooperatively, these processes replace the misspelled word: *lohng* with the word *John*.

9.1.3 Lexical Analyzer and Lexicon Organization. The lexicon is structured as a hash table. The lexicon organization is set up to facilitate spelling correction, phrase recognition, and abbreviations. There is no limit on the size of the lexicon. To test the system tolerance, I imported the Unix Dictionary, which has over 26,000 words. The system operated efficiently.

The phrase recognition algorithm recognizes all phrases contained in a string of words. For example, given the string: "*in addition to* " the system will recognizes the three words: *in*, *addition* , and *to*; then it also recognizes two phrases: *'in addition'* and *'in addition to'* . It interacts with spelling correction to use phrasal information to reduce the number of candidates when the error appears within a phrase.

9.1.4 Implementation Notes. The first version of this parser was produced in ‘C’. The second version is implemented in Common Lisp. The Lisp version strictly conforms to standard Common Lisp and does not use any system specific packages. We can confidently say it is highly portable. So far, the program runs on PC’s in *Allegro Common Lisp for MicroSoft Windows*; it runs on Sun Sparc Stations using *Carnegie Mellon University Common Lisp*, and it runs on Macintosh platforms using *Procyon Common Lisp*.

The system does not demand large amounts of memory or a fast CPU because of efficient use of time and space. The parser is table driven. A parser generator program, ALI, was created that takes as input a grammar, and generates LL(1) tables with augmentations of semantic rules in the grammar. The parser, in turn, is enhanced to execute the semantic rules as it parses the input.

To make it easy to add lexical entries, I have implemented a ‘C’ program, Lex_builder, that takes lexical entries as an input and generates the lexicon table, the word table and the hash table.

ALI and Lex_builder allow us to focus on the main research issues, and not to worry about generating the required tables. Both programs generate a very efficient layout for the parser and lexicon.

To allow incremental addition of words to the lexicon, the system allows the words to enter the lexicon with or without part of speech. Further, the semantic structure (lexical function) for the word can be left blank. The system inserts default lexical functions at run time for the entries that have no lexical function.

9.2 Future Study

9.2.1 Spelling Error Detection. We limited the detection of the erroneous words to only isolated words (words that do not exist in our lexicon). Some of the misspelled words, although only a small percentage, are garbled to a real word (an existing word in the lexicon). This type of spelling problem can be detected by using the sentence context.

9.2.2 Silent Characters. Another enhancement to my spelling corrector is to use lexical information about silent characters. When the system concludes that the character ‘c’ at location ‘m’ of the word from the lexicon is to be inserted at location ‘n’ of the misspelled word, the system needs to check the lexical entry of the word from the lexicon to see if ‘c’ is a silence character. There is no phonetic information in the current lexicon describing the sounds so the spelling corrector can take them into account.

Finally, similar sounding characters are not limited to a single character replacement. For example, if an ‘f’ is to be replaced with ‘ph’, or vice versa, lower error weights should be assigned as they sound similar. As an example consider the word *grafix* for *graphics*.

9.2.3 Word Segmentation. Separated words as well as joined words cannot be mixed with other types of errors, as they change the order of inputs. This is to say that while we can have up to a configurable number of ten different matches for a single word, we can only have one type of error when we split or join two input words. Similarly, when any of the replacement candidate words is a compound word [1], we take the following actions:

1. If the complex word is the best replacement candidate (the lowest error weight), just return the compound word.
2. Drop the compound word from the replacement list of candidates.

The spelling corrector looks forward for joining words and never backward. This causes problems in split words like

"specific ation" $\rightarrow$ *"specification"*

In this example, the word 'specific' is a valid word and we only realize we have a misspelled word when we get to 'ation'. Obviously, this is not difficult to do. On the surface, whenever there is an error, combine the misspelled word with the previous word and see if together they make up a word in the lexicon. It can get a little complicated when the leftover word (here *ation*) has error in it. For instance, when we have: *"specific taion"*, simply combining the two word does not produce a perfect word, and we are left with the question of whether to combine the two input words, or simply correct the misspelling of the second word.

The word boundary problem is considered for words that do not have any spelling errors except a missing/added blank space. In other words, we only consider the word segmentation when the only type of error is:

1. extra characters at the end of the erroneous word compared with the word from lexicon. This signals that there is a possibility of two words being joined. Now, we take the extra characters at the end of the misspelled word and search the lexicon for its existence. For instance if we have: *theincrease*, the program splits it into two words:

   the increase

   But for the entry: *theincreas,* (note that there is a misspelling in increase), the spelling corrector does not consider a word boundary error.

   Note that in our example, *theincreas*, where the first word does not have any spelling error, it is very easy to split the word, and then a simple spell correction on the second word, *increas*, takes care of the problem. I chose not to do this to save time. The system can handle this type of error if we are willing to allow some cpu time.

   This gets complicated when the first word has a spelling error, i.e. *tehincreas.* Here, there are two words that are joined and each one has a spelling error. The difficulty is to decide the location at which to split the word. Again, further study is needed to come up with a good strategy to handle this type of error.
2. extra characters at the end of the word from the lexicon compared with the erroneous word. This signals that there is a possibility of a word being split. We combine the misspelled word with the next input word if the resulting word matches the word from the lexicon. For instance, if we have: *incre* and *ase* the program combines them as: "increase". However, the two words:

---

[1] A word that must be replaced with a few words like: *cannot* which will be replaced with two words *can* and *not*.

incre and *a*es will be considered as two separate spelling errors, and the system returns replacement candidates for *incre* as:

increase, increased, increases,...

and for or *a*es:

yes, as, ...

I chose not to do this to save time. The system can handle this type of error.

The final caveat about word boundary errors is the limit of two words being joined or separated. At most two words are considered for either split or join. The spelling corrector only works when one word is split into two words, or two words are joined. For example, given the three words '*the increase in*' joined together as: *theincreasein*, the spelling corrector will not replace *theincreasein* with the three words *the increase in*. The problem is that the system detects that the word *the* is joined to another word, but since the leftover word *increasein* cannot be located in the lexicon, the spelling error rules out the word boundary problem. This is done despite the fact that the system can easily handle more than two words combined when there is no spelling error in any of the words. Note that the system does correct *increasein* to *increase, and in* when the respelling corrector is called to correct *increasein*. In our example, once the spelling corrector detects that the word ***the*** is joined with other words in ***the** increasein*, all it needs to do is to correct *increasein* to *increase, and in*. Instead, to save time, after taking the word ***the*** from *the* ***increasein***, the hash table is looked up for the word: ***increasein***

This can get complicated when the first word has spelling error. For example, in *hteincreasein* it is not obvious where the word boundaries are.

The reason for this shortcoming is a performance issue. The frequency of this type of error is very low. I concluded that it is better to fail on rare occasions rather than constantly slowing the system down. Note that when word segmentation problems are combined with regular spelling errors, a full search of the lexicon is usually required. With a large lexicon, this may not be feasible. This is because in an integrated natural language processing environment, such as Circsim-Tutor, the spelling corrector is not the only module of the system. Since this is a real time system, the user expects fast response. The overall efficiency of the parser is, therefore, critical.

9.2.4 Phrases. The phrase recognition algorithm presented in Chapter VII was designed to recognize the longest phrase. Looking for the maximum number of words in a phrase is not an error free approach. There may be a smaller phrase that is used. For example, in:

> *They committed several serious felonies before the police caught up with them. In addition, to escape, they robbed a bank.*

we actually have the phrase *In addition*, and not *In addition to* as the word *to* is used as part of the infinitive form of the verb *escape*. This situation involves complications not just with the following input word as in the phrase 'in addition' and the next word 'to'. We could have a problem with the proceeding word as in the phrase: *high blood pressure* where we could also have a phrase: *blood pressure* and the previous word *high* may not be part of the phrase. For instance,

> *I get motion sickness easily. Whenever I get up high, blood pressure problems result.*
>
> *He had multiple problems when he arrived to the hospital. He was drunk, high, blood pressure was also very low.*

The usage of a comma in the above examples simplifies the phrase recognition task. In fact, my phrase recognition algorithm considers punctuation marks and terminates the phrase at any punctuation. Unfortunately, many users typing a sentence to be processed by a machine do not bother to insert punctuation unless they are emphasizing a point. Also, there are situations where there is no need for a comma. For instance, consider the verb *mean*, and also the phrase *mean arterial pressure* in the following two examples:

> I measured mean arterial pressure.
> I mean arterial pressure.

In the first sentence, we have the phrase: *mean arterial pressure*. In the second sentence we have the verb *mean*, followed by *arterial pressure*. The difficulty is that mixture of the phrase: *mean arterial pressure* and the subphrase: *arterial pressure* stores different words in different locations of the chart. Table 9.1 shows that the three words: *mean arterial pressure* are saved as a phrase in location two of the chart. Table 9.2 shows that the word: *mean* is saved in location one of the chart, and the two words: *arterial pressure* are saved as a phrase in location two of the chart.

Table 9.1 Contents of the chart for: *I measured mean arterial pressure*

Word Number	Word String
0	I
1	measured
2	mean arterial pressure

Table 9.2 Contents of the chart for: *I mean arterial pressure*

Word Number	Word String
0	I
1	mean
2	arterial pressure

The phrase recognition algorithm must be extended to handle general cases. As we saw, there are complications in phrase recognition when a phrase contains subphrases. One solution to this problem is to allow multiple interpretations of various phrases. That is, to allow the lexical analyzer to have more than one interpretation for the next token and be able to mix phrases with simple entries. For example, if we have the word string *in addition to*, the lexical analyzer may return:

- (Phrase1: *in addition to*)
- ((Phrase2: i *n addition*) (single_word: *to*))
- ((single_word: *in*) (single_word: *addition*) (single_word: *to*)

The parser in turn must consider all the above possibilities and accept the one that does not violate the syntactic rules.

Another issue in handling phrases is the question of whether or not we need to save all forms of phrasal verbs like *go up*, or the system should be able to recognize *went up* as a phrase since *go up* is a phrase.

Currently, the system relies only on the existing phrases in the lexicon and makes no attempt to create derivatives of verbal phrases. This is a tradeoff between time and space. We have an entry in the word table for *went up* and the system could obviously recognize *went up* as a phrase. In contrast, we could save memory, at the expense of time by not saving the phrase 'went up'.

9.2.4.1 Solution to phrases with subphrases. To solve the problem of handling phrases that contain subphrases, as well as the mixture of separated/joined words with other types of spelling errors, one needs to change the input handling of the lexical analyzer.

One simple improvement to the lexical analyzer is to relax the condition of successive words in the input. The chart is a sequential repository of the input words. The word at location *n* is followed by the word at location *n+1*.

To relax the condition of successive words, each input word will have the explicit location of the next word, instead of the implicit next location. This allows us to mix a larger phrase and its smaller subphrases, as well as join/split words with other spelling error replacement candidates. For instance, when we have the following segment:

in addition to smoking ...

it could be organized in the chart as shown in Table 9.3.

Table 9.3 New Chart Organization

Index	Word Entry	Next Index
0	in	1
1	addition	3
2	in addition	3
3	to	5
4	in addition to	5
5	smoking	6

With this approach, there are three ways to analyze the three words: *in addition to* :

1. Take them as three separate words. In this case, the first entry is at location 0, and the next entries end at location 1 and 3, going to the next entries at location 5.
2. Take them as a phrase: *in addition* followed by word *to.* In this case, the first entry is at location 2, the next entry ends at location 3, pointing to the next entry at location 5.
3. Take them as a phrase: *in addition to*. In this case, the first entry ends at location 4, going to the next entry at location 5.

The challenge is to shield the parser from this representation. One solution is to group the input word entries. For instance, the group 0 entries are (0, 2, 4). Then, the group one entry starts at (1, 3, 5). Table 9.4 shows three ways to analyze: *in addition to* smoking.

Table 9.4 Grouping of a Phrase and Its Subphrases

Group	Word Entry	Word Entry	Word Entry
0	in	in addition	in addition to
1	addition	to	**smoking**
2	to	**smoking**	
3	**smoking**		

This change makes the lexical analyzer to work with two levels of input numbering, and the parser algorithm does not have to be changed/modified. The lexical analyzer does a mapping of the two levels.

9.2.5 Linkage of Lexicon Segments. Although the lexicon is segmented into 26 partitions, we search the lexicon sequentially, as a circular array, starting at the segment in which the first letter of the first word is the same as the first letter of the misspelled word. If a good replacement candidate is not found within this segment, we simply search the next sequential segment. If the segments are put in a dynamic linked list, considering adjacent characters, and similar sounding characters, can reduce the number of searched segments. Note that the links can be constructed statically. In other words, each segment will have a list of its next segments. The linked list can be modified for optical recognition, with respect to the shape of characters, i.e., next to the segment with initial character ‘O’ is the segment with the first letter ‘D’.

9.2.6 Optical Character Recognizer. In Chapter II we saw that the spelling error patterns to be considered depend on the application task. My spelling correction is designed to work with keyboard entry. If the system is used with an optical character recognizer (OCR), the table: ‘lex_adjacent_key_list’ must be modified. Currently this table contains adjacent character information for each keyboard character. For an OCR system, instead of adjacent character information, we would load this table with characters that look similar to the character.

Note that the table: ‘lex_adjacent_key_list’ does not have to have an entry for all items in this case. For instance, in an optical character recognizer, probably no character has features similar to the character ‘T’. Therefore there is no entry for ‘T’ in the table.

9.2.7 Speech Recognition. The use of context to narrow the choice of candidates can be extended for other systems such as speech recognition. Speech-to-text is often thought of as an alternative to the keyboard, particularly for those who have difficulty typing or for small systems without a full-sized keyboard [Meisel, 1993].

Speech recognition is still an imperfect technology [Roe, 1993]. As the accuracy of speech understanding continues to improve, sophisticated parsers like this one are needed to interpret the commands. Speech to text systems must deal with a wide range of homophones - words that sound the same but have different meanings [Crane and Rtischev, 1993]. With the ability of the parser to use context for spelling correction and disambiguation, this parser seems to be a good candidate to select the right word amongst the choices.

Kukich [1992a] mentions the Telecommunications Device for the Deaf (TDD) system developed at Bellcore, a first step toward making the telephone network accessible to hearing-impaired individuals. A TDD user can communicate with a voice phone user by calling a deaf relay center. The text typed by TDD users must be corrected before a speech synthesizer can produce intelligible speech. They have concluded that the greatest gain to be had in automatic spelling correction is the development of context-dependent techniques [Kukich, 1992a, p. 88].

The use of context is catching on (at least in a limited way) in industry [Rudnicky et al., 1994]. IBM's Personal Dictionary System (IPDS) is a dictation system, on PC's, that recognizes the user's words [Diehl 1994]. Malloy [1994, p.52] mentions:

> Since the task of recognizing individual words is so difficult, the IPDS uses context information to help decide what you are trying to say. IBM has developed a huge library of trigrams, that is three word combinations, that are common in English. With this information, the IPDS can choose not only among similarly sounding words, but also among homophones - words that sound exactly the same. It can, for example, correctly recognize the sentence: ' <u>They're</u> going to park <u>their</u> car there.

9.2.8 Syntactic Error Recovery. On the parser side, we have preserved the rich structure of LL, which can be used for syntax error recovery for the parser.

ALI is equipped to generate the necessary tables for *the Panic Error Recovery* method. Extending the parser algorithm to provide error recovery is straightforward. The challenge is that the addition of the syntactic error recovery adds another level of ambiguity to an already ambiguous environment.

Finally , this list is by no means complete. We expect that practical deployment of the parser will spawn off many new research areas. We hope that this research has provided the foundation for tackling them.

APPENDIX A

LEXICON BUILDER USER MANUAL

A.1 Introduction

As we saw in Chapter VI, the lexicon information is mainly spread into three tables: the hash table, the word table, and the lexicon table.

The actual tables in the lexicon are explained here. These structures are generated with default options of the lex_builder. Section A.5 provides a method for customizing the behavior of the lex_builder program.

Many arrays are used in my system. In fact not many lists are in use in the parser package. Accessing the n^{th} element of a vector is faster than finding the n^{th} element of a list, since the latter requires pointer chasing while the former requires simple addition. Vectors also take up less space than lists.

MacLachlan [1992, p. 76] states that arrays are often the most efficient representation for collections of objects because:

- Array representations are often the most compact. An array is always more compact than a list containing the same number of elements.
- Arrays allow fast constant-time access.
- Arrays are easily destructively modified, which can reduce consing.
- Array element types can be specialized, which reduces both overall size and consing.
- In comparison to a list representation, vectors (one dimensional arrays) are faster to access and take up between two and sixty-four times less space (depending on the element type.)

A.2 Hash Table

The hash table is called *lex_hash*. This is the main lexicon hash table. Given a word string it returns:

- Index to the word table slot that contains the string.
- NIL: If the word does not exist in the word table.

A.3 The Word Table

To deal with multiple entries in the lexicon table, and also work with abbreviations (i.e., bp for blood pressure), compound words (i.e., can't for can not), phrasal entries (i.e., "mean arterial pressure"), a table called word_table is used which contains:

1. Word string: The lexical entry's character string.
2. Index range to the lexicon table which is a pair: <low, high>. As we saw earlier, a word can have multiple entries in the lexicon. For instance, the word *increase* has two senses as a verb (go up, and cause to go up). It also has two senses as a noun. Therefore, there are four entries in the lexicon table for *increase*.

   If the word is a meaningless word, then there is no index to the lexicon. For instance, the phrase 'vice versa' contains two meaningless words: 'vice' and 'versa' that must be in the word table but do not have any entry in the lexicon table.

3. Abbreviation: an index to the word table for the unabbreviated word. For instance, *bp* has an index to the location of *blood pressure* in the word table.
4. Compound entry: an index to the compound table that contains the actual replacement string.
5. Phrasal flag: This indicates whether or not the word is part of a phrase.

The word table is mostly used by the parser to deal with classification of the user input. Note that an entry may have many of the above slots. For example, 'i' is a word by itself and is also an abbreviation for the word 'increase'.

The last three entries: abbreviation, compound entry, and phrasal flag are very sparse. To save space, I split up the word table in six different tables.

A.3.1 word_table_symbols. Contains the word string component of the word table. This is the full array of lexical entries. The spelling corrector accesses this table frequently. It is beneficial to have the character string of the word in an array by itself in a paging system.

A.3.2 word_table_low_indexes. Contains the low range of indexes to the lexicon table for this string.

A.3.3 word_table_high_indexes. Contains the high range of indexes to the lexicon table for this string.

A.3.4 word_table_compound_indexes. Contains the actual text of the word that the compound word must be expanded to. Most lexical entries are not compound words.

In order to save space and deal with the fact that the compound entries of the word table are very sparse, we use either a hash table or an association table to save only entries that have actual compound words.

For example if the hash table returns index *n* for the word *can't*, then there is an entry: (*n*, (can not)) in this table.

A detailed discussion of the usage of the hash table and the association list follows in later sections. The default setting is a hash table.

A.3.5 word_table_abbreviated_indexes. This table contains indexes to the word table for the unabbreviated word. For example if 'bp' is at location 234 of word table, the abbreviated field has index 567, where 567 is the index to the word table that contains the unabbreviated words: "blood pressure"

Most lexical entries are not an abbreviation of another word. As with the compound words we use a sparse matrix to deal with the problem.

The word_table_abbreviated_indexes table contains the associations of the word table index (of the abbreviated word) and the index to the word table for the unabbreviated word. For example, if the hash table returns index 234 for the word 'bp', then there is an entry: (234, 567) where 567 is the index to the word table that contains the unabbreviated words: "blood pressure".

A.3.6 word_table_phrase_flags. This table contains phrasal flags. The words that are part of a phrase have this flag set. We distinguish between words that are the first word of a phrase, or second, third, ... words. Two flags are saved:

1. initial_phrase: When set this flag indicates that this word appears as the initial word in a phrase.
2. phrasal flag: When set this flag indicates that this word appears in a phrase.

Most lexical entries are not part of a phrase. As with the compound words, we use a sparse matrix to deal with the problem. For instance at the time of writing, we only have 68 words out of 1621 lexical entries that are part of a phrase. The entries here are of the form: <word table index, flag value>. Needless to say, there is no need to save the words that have the flag value zero. When there is no entry for index ‘i’ in this table, that indicates that the word at location ‘i’ of the word table is not part of any phrase.

A.3.7 Hash or Association List for Sparse Tables. These sparse tables are stored in a table that contains the word table index and associated data: (*index, data*). By default, I use a hash table to save the entries. The index is used as the key to the table. If we are looking for the data item in location *n* of the table, then *n* is given to the hash table and *data* is retrieved. An alternative to hash table generation is to use an association list. Obviously, a hash table for small size arrays is not a good idea. As mentioned, an association list would do. Having a large lexicon in mind, I use a hash table. If the hash table is to be replaced with an association list (only if we are dealing with a small lexicon), then the access function in the file: *lexicon.lsp* must be modified. Note though, in a small lexicon all methods work anyway. It is the large lexicon that we should be concerned with. Although the use of a hash table may be overkill for a small lexicon, the use of an association list for a large lexicon is not acceptable when there are hundreds of entries for the items (consider a lexicon of 50,000 entries, then we may have 500 abbreviated entries). If your association list has more than about 10 entries in it, consider using a hash table. Hash tables are often more efficient.

The following is copied from General Programming Style Rules in net news: comp.lang.lisp:

- Both association lists (alists) and hash tables may be used to represent tabular data. Hash tables have an O(1) running time and alists an O(n) running time, so hash tables are ultimately more efficient than alists. However, if the alists are small, they can be more efficient than hash tables, which have a large initial overhead.
- Alists can sometimes be more efficient if the keys are sorted according to frequency, with the most heavily accessed keys appearing at the front of the list. But one doesn’t always know this kind of information, and even then the frequency distribution may be flat.
- In Allegro CL 4.1 [SPARC; R1], the rule of thumb is that for less than 24 elements, linear search using alists beats hashing. In Lucid CL 4.0.1 HP 9000/700, the break-even point is at 10 elements. The break-even points vary in other lisps from as low as 4 elements to as high as 100 elements.

So if you're using alists in your code, using hash tables instead may speed up your program.

A.4 The Lexicon Table

The lexicon table has the following data:

- pos: the part of speech of the words.
- index2wordTable: Backward index to the word table. This is not actually needed. We will dump it if it turns out to be useless. I used it for a sanity check of the lexicon.
- structure: The word semantic structure. A detailed discussion of the semantic structure follows.

The format of the third item (structure) is not finalized yet. So far we have the following information:

- LFG functional schemata: at this point the structure is actually an a-list such as the entry for the word "sees" :
  ((PRED \"see\" SUBJ OBJ) (SUBJ ((NUM SING) (PERS 3))))
- Case Frame for Verb Information.

This information is shown graphically in Figure A.1.

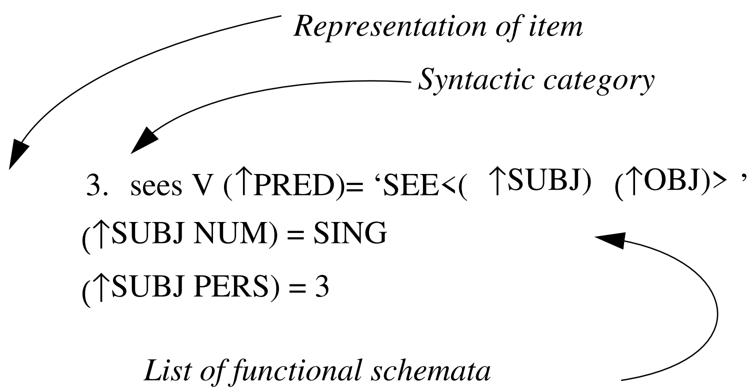


Figure A.1 Format of LFG Lexical Items

The semantic data is recorded in a Lisp structure, which is called a '*defstruc* ture' in Lisp terminology [Steele, 1990]. Since words with different parts of speech have different properties, each part of speech has its own structure that contains the lexical properties of the word. For instance, for a '*noun*' we need to know if it is *singular* or *plural*, *count* or *mass*, whereas for a '*verb*' we want to know its *tense* and other material relevant to verbs. This is done so each one of them has the minimum number of fields needed.

Since the system knowledge base is written in CLOS, we considered the possibility of saving structures as objects (in CLOS) instead of Lisp structures. Since the lexicon is actually a data repository, I decided to use a simple structure without the overhead of CLOS. The General Programming Style Rules from *comp.lang.lisp* from the netnews states:

> When you don't need the full power of CLOS, consider using structures instead. They are often faster, take up less space, and are easier to use.

To avoid accessing multiple dimensional arrays, the lexicon table is split into three tables: lexicon_pos, lexicon_index2wordTable, and lexicon_structure.

A.5 Customizing the Lexicon

The lexicon is a very dynamic organization. As words are added or removed, certain conditions may not remain true. For example, at this point, abbreviated entries are very sparse. In a certain environment, this may not be true. The system architect of a new lexicon may decide not to use sparse matrixes. Or, instead of using a hash table for sparse matrices in the current lexicon, (s)he may want to use an A-list. With this in mind, I made sure that the lex_builder program is customizable.

The task of building a lexicon was very dynamic. Obviously, this structure was not created overnight. I deployed this structure with two factors in mind: 1) fast access, and 2) efficient memory usage.

After the initial word table was built, I noticed that many slots are sparse. Instead of hard coding the program to generate the above structure, I have designed the lexicon builder to be customizable for different environments.

The behavior of the lexicon builder is controlled through setting the C preprocessor switches. Unfortunately, the program got more complicated than I anticipated. So, only attempt to modify the switches if you are a very experienced C programmer, and more important, you know exactly how you want the lexicon structure to look.

Needless to say, if the lexical structure is changed, the access functions in the file: *lexicon.lsp* must be updated.

I have introduced the concept of *virtual phrase* to ease recognition of phrases that are more than two words long. A virtual phrase is a phrase with no entry in the lexical table. The CREATE_VIRTUAL_PHRASE switch controls building or not creating virtual phrases. Note that this is an important issue. My phrase recognition algorithm requires the existence of the virtual phrases in the lexicon. Change this flag if you change the phrase recognition algorithm.

The symbol field of the word table is printed in a one dimensional array. The rest of the indexes could be in one six dimensional array or in six different arrays. The switch SINGLE_ARRAYS_OF_ITEMS controls this, if it is defined we put every index in a single array. If you turn the switch off, then the word table will not be split.

Some of the word table components: compound, abbreviated, initial phrase flag and phrase flag are sparse. Most entries are null. A way to save space is to use a sparse matrix (index value) for the items with no missing values. We use a hash table to speed up the search. The variable:

USE_HASH_TABLE_FOR_SPARSE_ITEMS

controls this process. If it is defined then we use hash tables. If this switch is not defined, then regular arrays are used and all items are printed in them. The variable is effective only if SINGLE_ARRAYS_OF_ITEMS is defined.

To save space in the hash table, we could combine the two flags, the initial phrase and the phrase flag as two binary numbers in the same hash table. The switch: COMBINE_PHRASE_FLAGS_OUTPUT

when defined, allows the combination of the two flags as shown in Table A.1.

Table A.1 Binary Values of Phrasal Flag

Initial Flag	Phrasal Flag	Binary Number	Comment
0	0	0	None, the word does not appear in any phrase.
0	1	1	The word appears in a phrase.
1	0	2	The word appears as the initial word of a phrase.
1	1	3	The word appears in at least one phrase as its initial word, and also in another phrase (but not as the initial word).

Both of these two flags are combined as a binary number in the table: word_table_phrase_flags.

The contents of the *lexicon_structure* can be saved in the ‘lex_table.lsp’ file as a string of symbols. The variable: PRINT_IT_AS_STRING controls it. Currently, this switch is undefined, and the value is saved as lists. To save it as a string, you need to define this variable.

The switch PRINT_SEPARATE_COMPOUND_TABLE, when defined, causes the creation of an array: compound_word_symbols. The hash table: word_table_compound_indexes will have an index to the compound_word_symbols. When not defined, the list of words are directly in the table:

word_table_compound_indexes.

A.6 Lex Builder Process

The lexicon is made in two steps. Step one: raw lexical entries are processed. Lexical entries are input from a file in the format:

a (pos art definite no)
a-m orientation (pos noun abbrev-for “actin-myosin orientation”)
abbreviation (pos noun cm count num sing gender neuter)
abbreviations (pos noun cm count num plural gender neuter)

The raw lexical entries are processed and saved in the file: *lex_tables.c* by the first step. The table: lexicon_words is created with the following format:

- Lexical entry word string
- Lexical entry part of speech
- Lexical entry structure.

The second step generates the Lisp tables to be loaded in the Lisp environment. A detailed report is generated in file "/tmp/file.out" which displays the content of the word table and the lexicon table in a readable format.

A.7 Lexical Access Functions

A set of functions are provided for accessing the lexicon. The functions are intended to hide the lexicon structure from the user. I have just a few samples of available functions in this section. For the complete list please look at the file: lexicon.lsp.

The idea behind this file is that the builders of the rest of the system should not worry about the lexicon structure. In fact, one of the components of the lexicon was a table 'word_table', which contains various information. After the table was actually built, I realized that there are many sparse entries there. The table was split into six other components, without affecting the parser. Not a single line of the parser code had to be changed.

The main access function is:

LEX_word_table_index_off_hash(string) which returns the index to the word table for the given string. If the string is not in the hash table the function returns NIL.

The following functions extract the word table contents:

getWordTable_symbol(*index*): Returns the word string at location *index* of the word table.

getWordTable_lexIndex_low(*index*): Returns the word string at location *index* of the word table. If the return value of this function is nil, it implies that the word does not have an entry in the lexicon table.

getWordTable_lexIndex_high(*index*): Returns the first (*low*) index to the lexicon table for the entry at location *index* of the word table. If the return value of this function is nil, it implies that the word does not have an entry in the lexicon table.

getWordTable_abbrevIndex(*index*): Returns the last (*high*) index to the lexicon table for the entry at location *index* of the word table.

getWordTable_compound(*index*): Returns the replacement words for the compound word at location *index* of the word table. If the word at location *index* of the word table is not a compound word, returns *nil* .

getWordTable_initFlag(*index*): Returns *t* if the word at location *index* of the word table appears as the initial word of at least one phrase in the lexicon, *nil* otherwise.

getWordTable_flag(*index*) Returns *t* if the word at location *index* of the word table appears in at least one phrase (but not the initial word) in the lexicon, *nil* otherwise.

The above set of functions combined with the hash function:

LEX_word_table_index_off_hash()

give complete access to the word table. Note that the hash function returns the word table index.

The following functions provide access to the lexicon table:

LEX_lexicon_pos (): Takes an argument, index (int number), returns the part of speech of the word at location 'i' of the lexicon.

LEX_lexicon_structure() : Takes an argument, index (int number), returns the structure associated with the entry at location 'i' of the lexicon.

LEX_lexicon_symbol : Takes an argument, index (a number), and returns the word entry (character string) at location 'i' of the lexicon table. The lexicon actually does not contain the word symbol, we use its index to the word table to return its character string.

Note: to get all senses of a word *w* requires the following steps:

1. *index* = *LEX_word_table_index_off_hash(* w).
2. *low* = getWordTable_lexIndex_low(*index*)
3. *high* = getWordTable_lexIndex_high(*index*).
4. Now look at the lexical entries between the two indexes: < *low, high* >, to get all senses of the word: *w*.

Many convenient functions are provided. A small sample of them are listed below:

getLexIndexOffHash4pos() Takes two arguments, the word string (*str*) and the desired part of speech (*pos*), accesses the hash table and gets the word table index for *str*. Then it starts from the low index and loops through the high index in the lexicon and returns the lexicon index with the given *pos*. If the *str* is not in the hash, or the word table does not have any index to the lexicon for *str*, then it returns NULL. For instance:

getLexIndexOffHash4pos("artery", noun) returns the lexicon index for "artery".

getLexStructureOffHash4pos() Takes two arguments, the word string (*str*) and the desired part of speech (*pos*), passes them to: *getLexIndexOffHash4pos* and then returns the lexicon structure at that index.

lexicon_report( ) Prints a report on lexicon entries. Information such as abbreviated words, compound words, words with multiple parts of speech and phrases are printed out.

test_lexical_entries () Checks the correctness of three tables: lex_hash, word table and the lexicon table. Starts with word table, for every entry accesses the hash table to make sure we have the entry in the hash with the right index.

Some of the functions should be extended further or additional functions must be added. For example getLexStructureOffHash4pos returns the structure of a word with a given part of speech. Later, we may want to access slot '*x*' of the structure. It will be a minor job to have another function that takes two arguments: the Lisp structure and the slot name in the structure, and return the content of the slot within the structure.

APPENDIX B

THE PARSER INTERFACE

The parser operation can be viewed using the following two debugging switches:

- PRS_DEBUG
- LFG_DEBUG

Debugging switches are mostly used to report the progress of the parser. It is important to have debug switches as a constant variable, so, as we compile the files, the sections of code that use these switches are simply deleted when the switch is NIL. CMU Lisp does this very efficiently.

The main access function for the parser is the function ‘parse’. It can be called with three type of arguments:

1. String: (parse “cardiac contractility”)
2. List of tokenized strings: (parse ’(“cardiac” “contractility”))
3. List of symbols (parse ’(cardiac contractility))

All three calls result in the following structure:

```
(PARSE_RESULT [1]

    :STATUS FULL_SENTENCE
    :F_STRUCTURE
    ((PRED “ANSWER” ANS) (ANS ((PRED “*cc*”) (PAR +))))
     :C_STRUCTURE
    #S(CS_NONTERMINAL
    :SYMBOL “S”
    :PROD 9
    :RHS (#S(CS_NONTERMINAL
             :SYMBOL “ANS”
            :PROD 39
            :RHS (#S(CS_NONTERMINAL
            :SYMBOL “PAR”
            :PROD 0
                    :RHS (#S(CS_TERMINAL
                    :SYMBOL “CARDIAC CONTRACTILITY”
                    :POS “noun”
```

[1] This is a full sentence using answer grammar described in Section 8.3 on page 157.

:MATCHING “cardiac contractility”)))))))

:FAILURE_REASON “ ”

:TEXT “ ”

:MISSPELLED_COUNT 0)

To examine the result, check the slot F_STRUCTURE. If it is not nil, it indicates that parsing succeeded and the f-structure was successfully built.

If the f_structure is nil, it indicates that the parser was not able to build the f-structure. In this case you need to examine the rest of the information. First check the flag *status* for the parsing report. If it is nil, parsing failed.

If the *status* is set to a fragment, then the input was a fragment; parsing succeeded.

If the *status* is set to full_sentence, then input was a full sentence and the parsing succeeded.

When the f-structure is not constructed, the parser tries to explain the problem. If there were any spelling errors, they are blamed. In this case, the field failure_reason contains the message:

Sorry, I do not understand the word < misspelled word >

and the text field lists the misspelled words. If there is no spelling error, we issue the message:

Sorry, I do not understand the sentence: < input sentence >

and the text field contains the input sentence. The input sentence displayed in the text field will be the same as the given sentence to the parse function.

Note: when the slot *f_structure* is not *nil*, the sentence is successfully parsed (either as a fragment or full_sentence) the c-structure field contains the c-structure, and the f-structure field contains the f-structure.

I have added a function: decode_parse_result that prints out a detailed description of the *parse* function, as well as a detailed report of the spelling errors and replacement candidates.

A few other useful functions are provided:

1. prs_summary: prints the constructed f_structure and the c_structure.
2. lex_analysisOfAwordList: Calls the lexical analyzer for the argument list. The argument list must be a list of tokens, i.e., *(“i” “saw” “a” ...)* or a simple string, i.e., “ *I saw a man ...* ”. Naturally, we could have spelling errors in the list, and the system corrects them..
3. show_chart: displays (briefly) of the content of the chart.
4. show_tree: prints the syntax tree. This is useful when we have failed to build f_structure so prs_summary may not print anything.
5. info_out: prints details of the chart and all constructed syntax trees in great detail.
6. A lot of printing functions are provided in file prs_out.lsp.

APPENDIX C

MAKING THE PARSER

To make the parser package, one needs to define the grammar and the lexicon. A sample input file to ALI is provided in Section 4.5 on page 73. This file must be named 'gram.ali' in the subdirectory: gram. If you like to use other names, simply modify the makefile in the subdirectory.

The parser generator and the lex_builder were written in C. Both have been tested on the PC and the Sun machine, but not on the Macintosh. Theoretically, they should work in a Macintosh, but I have not tested them, since I had no access to a C compiler in a Mac. However, the parser package has been tested under *Procyon Common Lisp* on the Macintosh platform. To run the system on a Macintosh, the parse table and the lexicon must be constructed on a PC or a Sun machine.

Issue the command **make** and in a few minutes the system is ready. The make command works in both Sun machines and PCs. In both platforms, the grammar tables and the lexicon tables are generated automatically. In the Sun machine, the generated grammar and the lexicon are compiled automatically by the Lisp compiler. In the PC you have to do the Lisp compilation manually, because the Allegro Lisp cannot be called in the make file. This is not a big deal. Get into the lisp, and compile two files: *gram_out.lsp* and *lex_table.lsp* .

If you need to compile all lisp files, in the subdirectory: *prs*, there is a file called: *compile.lsp* (and *compile.pc* for the PC version). If you load this file into the Lisp system, all parser files are compiled.

C.1 Grammar Changes

Changes in the grammar must be made in file *gram.ali* . A sample file is given in Section 8.2.1 on page 156.

C.2 Lexicon Changes

Changes in the lexical entries must be made in file *lexicon.c* . Each lexical entry has three entries in the file:

"word string", "part of speech string", "lexical function string"

Two levels of testing are performed on the lexical entries. First, your change will be compiled using the C compiler. The most common error here is to put an end of line in any one of the strings. If your entry is long and does not fit in a line, before the end of line character put the escape character '\', (before pushing carriage return, enter '\').

The second level of testing is performed when the generated Lisp files are loaded (the Lisp file is generated by lex-builder program). Most attention in this test is given to correctness of the lexical function entries.

C.3 A Word of Advice

Every effort was made to make the system efficient. This parser performs a lot of tasks including context dependent spelling correction, which has made the package complicated. It is essential to have all parser files compiled under Lisp and not run them in the raw text mode (interpreted). The system will not perform well in interpreted mode.

At this point, the advantages of compiling code relative to interpreting it probably does need not to be emphasized, but remember that in most Common Lisp systems, compiled code typically runs hundreds of times faster than interpreted code. Also, compiled files load significantly faster than source files, so it is worthwhile compiling files that are loaded many times, even if the speed of the functions in the file is unimportant [MacLachlan, 1992, p. 82]

There are many *debugging* switches in the system that are defined as constants. When the debugging switches are turned off, the compiler actually throws away the corresponding code. This has two side effects:

1. A lot of warnings are issued by *Carnegie Mellon University Common Lisp*, stating that unreachable code is being deleted.
2. Execution is faster as the process does not have to execute the condition.
3. The executable code is smaller as the debugging sections are thrown away.

APPENDIX D

SYSTEM VARIABLES

D.1 Spelling Corrector Variables

The spelling error process must be tuned for your environment. The default values are selected for the method to be generic. Leave them as they are if you do not have performance problems. The rule of thumb is that the lower these values, the faster the program works. The larger the values, the wider the range of correction. The size of the lexicon plays an important role here.

Whatever values the variables are set to, please preserve the relationships between them. For instance, do not use a smaller number for *maximum disagreement* than *average disagreement*.

If the corresponding relation is not respected, the program may not operate properly. I have plan to add a function for sanity check of the variables in the spelling error module.

Error weights and the disagreement limit and the maximum number of selected candidates are all configurable. These variables affect the operation of the spelling corrector.

Function *display_spell_variables* presents the value of the variables that control the spelling correction operation. These variables are to be changed to constant once the practical usage of system has determined good values for them. Error weights are numbers between 0-100. Note zero means no error. The following is the result of running the function: d*isplay_spell_variables*:

Character case mismatch:	*3*
Reverse character mismatch:	*60*
Missing character:	*80*
Missing character at the end:	*12*
Added character mismatch:	*80*
Substitute character mismatch:	*70*
Adjacent character mismatch:	*45*
Similar sounding character mismatch:	*45*
Acceptable disagreement:	*12*
Maximum % of disagreement:	*45*
Average disagreement:	*30*
Size of Spell list:	*15*

The most important value is the *'Size of Spell list'*. The larger it is, the better the chance to find better candidates. If it is set to one, all advantage of my system is lost, as the spelling corrector will return one replacement candidate for the misspelled word. A reasonable range for this number is in the range (5, 20). The optimal value depends on the system, and how much time we are prepared to give to the spelling corrector.

The value of '*Maximum % of disagreement*' also affects the spelling corrector operation. This variable acts as a filter. If the number is too high, we get a lot of irrelevant words. If it is too low, a lot of good candidates are discarded. The higher tolerance for error can affect system efficiency as the comparisons would not be terminated in early stages. A reasonable range for this number is in the range (35, 55). While setting this number make sure that the value of *'average disagreement'* is set accordingly, it must of course be smaller than the value of '*Maximum % of disagreement*'. Note that the system uses an adaptive approach, see Section 5.3.1 on page 77, to lower the value of '*Maximum % of disagreement*'.

The rest of the variables, play important roles too. For instance, the variable that controls *'average disagreement'* can play a significant role. As soon as a word is found with the disagreement less than the average, we set the disagreement limit to the value of *'average disagreement'*. Care must be taken to set the average value, as it may exclude many words if we find a word with small disagreement with the misspelled word. For example, lets assume that we have the erroneous word: 'mae'. This word matches 'map' with one replacement, and matches 'mean' with two corrections. If we compare it with 'map' first, we may exclude accepting 'mean' as possible candidate if the average value is smaller than the error weight for correcting 'mae' to 'mean'. On the other hand, setting the average disagreement to a high value slows the system down.

The variable *!allow_4th_wayCharComparison!* controls the method of comparison. If it is defined (not null) 4 way match is performed, otherwise three way match will be done.

D.1.1 Compound Word Handling. The spelling corrector finds a few replacement words for the misspelled word. One or more of the replacement words can be a compound word. Since a compound word needs to be replaced with two or more words, compound word replacements cannot be mixed with regular replacement. When the first choice for replacement is a compound word, we drop the rest of the replacement candidates and the spelling corrector simply returns one replacement word. When the compound word is not the first choice, we drop the compound replacement word.

The difficulty with mixing the compound word and regular words is handling the variable number of input words. The same solution presented for handling of phrases that contain subphrases can be used here. The details of the solution is in Section 9.2.4.1 on page 178.

D.2 LFG Operation Control.

The variable: require_PRED controls behavior of analysis of f_structures for Coherence problem. The variable: require_PRED will require that all lexical entries have a PRED in their functional schemata, when it is set to TRUE. When require_PRED is set to NIL, we quietly allow the absence of semantic forms.

D.3 LFG Function Activation

Activation of the grammar actions is controlled by the variable: !GRAM_ACTION_FUNC_FIRING!. We fire actions if this variable is *true*. When it is null, then actions are not activated. Note the variable:

!DELAY_LFG_ACTION_FUNC_FIRING!

controls the behavior of the lfg action functions. If the variable is *true*, they do not perform the action and return TRUE.

D.4 Wild Card Part of Speech

The system provides a mechanism for wild carding of words with *unknown* part of speech. To have a flexible lex_builder, we allow entries of words with no part of speech and no semantic structure. This creates flexibility for spelling correction. At the same time, when a word is needed in the lexicon, we can enter it with no definition and add the properties later.

The variable: !PRS_ALLOW_WILD_CARD! controls the operation. If it is nil, the lexical analyzer throws away all words with *unknown* part of speech. In this case, the system basically does not work when a word with an *unknown* part of speech is in a sentence to be parsed.

There are two levels of control for wild carding. One level of control is located at the system level, which is controlled by the variable:

!PRS_ALLOW_WILD_CARD!

The second one controls the operation of the spelling corrector. The spelling corrector ignores the words with *unknown* part of speech when the variable: !SPL_ALLOW_WILD_CARD! is set to nil.

Note that if the wild carding is not allowed, then spelling correction for phrasal entries may not work if any word in the phrase by itself does not have a part of speech. This is a serious problem since many words in a phrase may not have meaning, i.e. in *vice versa*, what does ‘versa' mean? It is much better to allow SPL_ALLOW_WILD_CARD to be set to true. If the system is not to have wild cards, control it by: !PRS_ALLOW_WILD_CARD! variable. Note that variable !PRS_ALLOW_WILD_CARD! controls the overall behavior of wild cards. If it is nil !SPL_ALLOW_WILD_CARD! will not change the final outcome.

APPENDIX E

THE SYSTEM FILES

The system files are organized in three directories:

1. C: C version of the parser.
2. Lisp: Contains the Common Lisp version of the parser.
3. Tools: Various programs and tools to generate test files and data for the thesis.

The following sections present a description of the files in each category and a brief comment about the file contents. Note that we are limited to eight character file names for DOS systems on the PC.

E.1 The Lisp Directory

The files containing the Lisp version are organized in several subdirectories. The Lisp version is more complete than the C version, as the LFG related programs are only implemented in Lisp.

E.1.1 Subdirectory: ali.

Another Language Interpreter files. This is the parser generator tool.

ali.c	Various support functions for the tool.
ali_dec.h	Declarations and structures.
gram_out.c	Print facilities for the output grammar.
message.h	Predefined messages for the outputs. Includes support functions for the parser.
strings.c	All string manipulation functions.
ali.h	Ali’s internal data structures.
ali_ll1.c	LL(1) Analysis and table generator.
platform.h	Platform specific definitions, for Sun and PC machines.
ali_dec.c	All global variables.
ali_main.c	Ali’s main program. Takes care of system files
makefile	Make file

E.1.2 Subdirectory: bin.

This is the executable directory for the Lisp version of the parser.

E.1.3 Subdirectory: gram.

The grammar directory contains various grammar files.

connie_gram.ali	Circsim Tutor Version 2 grammar
answer.ali	Answer grammar.
makefile	
gram_tom	Tomita’s grammar
simplified.tomita	Simplified Tomita’s grammar.
lfg.ali	A small grammar (Michael Wescoat) for testing the LFG support in the parser.

E.1.4 Subdirectory: gram/lisp. The Lisp related directory for the grammar. The files here actually translate the output file generated by ALI from C to Lisp.

gen_lisp.h	Header file
gram_lsp.c	Main program that does the generation.

E.1.5 Subdirectory: include. Header files for the global headers used in the C version.

E.1.6 Subdirectory: lexicon. Lexicon related files directory. The subdirectory, builds the lexicon tables to be loaded in Lisp.

build_hash.c	Generates the tables for the lexicon. This files uses the result of the first phase of lexicon builder.

E.1.7 Subdirectory: autoload. Automatic loading of the raw lexical entries. Contains files that make lex_builder program.

general.h	General header file for lex_builder program
grammar_rules	Grammar description for the raw lexical entries.
parser.c	Generated parser for the grammar rules
parser.h	Parser header file.
lex.h	Lexical header for the lex_builder program

E.1.8 Subdirectory: lib. Library files, generated by various makefiles in the Lisp directory. These are intermediate files for the parser (lexicon library file, grammar library files, ...)

E.1.9 Subdirectory: prs. Parser related files. All files automatically generated by ALI and lex_builder are put in this directory. In fact, this directory holds all of the system files to be loaded into Lisp.

OLD.gram_out.lsp	Old copy of parse tables. Every time a new parse table is installed, the existing file in this directory is renamed to *OLD.gram_out.lsp* , in *Unix* system (this is an illegal file name in *DOS*).
OLD.lex_table.lsp	Old copy of lexicon tables. Every time a new lexicon table is installed, the existing file in this directory is renamed to *OLD.lex_table.lsp* , in *Unix* system (this is an illegal file name in *DOS*).
chart_pr.lsp	Chart access functions.
compile.lsp	Compilation make file.
counter.lsp	Counts number of parse tree generated for a given input. The functions in this file generated data for Table 3.6on page 61.
gram_out.lsp	Parser table. Final grammar files generated by ALI.
gram_prt.lsp	Various functions to print grammar related tables and data.
keyboard.lsp	Keyboard information data such as adjacent keys and similar sounding characters are kept in this file.
lexicon.lsp	All access functions to the lexicon are kept in this file. Outside of this file, there should not be any access to tables in the file: ‘lex_table.lsp’. The internal structure of the lexicon is hidden from the

user, and a few user friendly functions are provided here that satisfy all needs for access to the lexicon.

lfg_act.lsp — All LFG grammar schemata are enforced in this file. (All LFG related grammar actions.)

lfg_basic.lsp — Basic LFG functions to construct the f-structure

lfg_data.lsp — LFG internal data structure.

lfg_cnst.lsp — LFG consistency check.

lfg_gen.lsp — LFG c and F structure generator.

lfg_prnt.lsp — Various print out functions for lfg c structure.

lfg_prs.lsp — LFG parser related routine. Calls the parser and after successful parsing of the input, calls LFG functions in lfg_gen.lsp to construct C/F structures.

lfg_spell.lsp — Spelling corrector interaction with the process of LFG f-structure function build-up.

load.lsp — Loads the parser files into the lisp

makefile — Make file.

phrase.lsp — Phrase recognition functions.

prs.lsp — Parser related functions.

prs_data.lsp — Internal data structure of parser

prs_misc.lsp — Miscellaneous parsing functions. Mostly for time measurement and testing.

prs_out.lsp — Many functions are provided here that display the chart with various details.

prs_str.lsp — String manipulation related functions.

prs_test.lsp — Parser test functions. Using this file, I was able to apply the parser to the tutor files to parse student inputs.

prs_var.lsp — Parser internal variables.

shortcut.lsp — A small file that gives keyboard short cuts for the most used functions. The function names in general are long and descriptive.

lex_spel.lsp — Main functions of spelling corrector

support.lsp — Various supporting functions for the parser. These are mostly small function that perform specific task such as insertion sort of an item into a list.

suspend.lsp — provides functions to suspend/resume productions, to handle left recursive (direct/indirect) nonterminals..

test.lsp — A few functions for sanity checks, such as testing for hardware dependencies and phrase recognition.

token.lsp — String tokenization function.

E.2 The C Directory

This directory contains the C version files organized in a few subdirectories.

E.2.1 Subdirectory: ali. Another Language Interpreter files. This is basically the same structure as in directory Lisp/ali described in Section E.1.1 on page 196.

E.2.2 Subdirectory: bin. Executable directory for the C version of the parser.

ali	Ali's executable file
chart_parser	Parser executable.

E.2.3 Subdirectory: bin/ analyze. Analysis of the output. These files were used to generate data used in Figure 3.10, on page 64.

E.2.4 Subdirectory: gen_test. A random sentence generator for a given grammar.

gen.c	Generates random sentences.
gram.c	Copy of the grammar (output from ali)
gram_lex_out.c	Gives report of the grammar and the lexical entries.
makefile	

E.2.5 Subdirectory: gram. The grammar directory contains various grammar description files.

connie_gram.ali	Circsim Tutor Version 2 grammar
answer.ali	Answer grammar.
makefile	
gram_tom	Tomita's grammar
wescaot	Michael Wescoat's grammar

E.2.6 Subdirectory: include. Header files for global headers used in the C version.

E.2.7 Subdirectory: lexicon. Lexicon related files directory.

lex_search.c	All search functions. In addition to the hash search, the two binary searches are included in this file. The report given in Section 6.10 on page 136 for the binary search is generated from the code in this file.
prime.c	A prime number generator
Lex.l	Lexical analyzer rules written in LEX.
lex_spell.c	Spelling corrector program.
build_hash.c	Hash table generator
lex_tables.c	Lexical entries in raw form.
hash_temp.c	Data structure generator for the hash table.
makefile	Lexicon makefile.
test_hash.c	A few functions that test the correctness of the hash table and measure the access time. The data in Section 6.10 on page 136 were produced using this file.
lex.h	Header files.

E.2.8 Subdirectory: lexicon/build_lex This directory contains functions that I used to examine a number of Circsim-Tutor keyboard to keyboard transcripts and extract about the 4000 words used in our sublanguage. The generated words are used for the current lexicon.

E.2.9 Subdirectory: lib. Library files, generated by various makefiles in C directory. These are intermediate files for the parser (lexicon library file, grammar library files, ...)

E.2.10 Subdirectory: prs The parser file directory, contains all parsing related files.

check.c	Checks the consistency of the pointers in the chart.
prs_dec.c	Global variables used in the parser.
prs_mem.c	Dynamic memory related functions, allocation and deallocation.
prs.c	Parser main routines.
prs_inp.c	Parser input related routines.
sets.c	Supporting functions for set operations.
prs.h	Parser header files.
prs_lisp.c	Generates a Lisp like list from the chart.
prs_priv.h	Parser private header file.
prs_actions.c	Handles the grammar action functions
prs_lrec.c	Handles left recursive rules.
prs_prnt.c	Output generator for the result of the parser.

E.3 Tools

Various programs that were used to generate data to support the thesis.

E.3.1 Subdirectory: LR_states. This directory contains the modified version of YACC rules and other functions that were used to generate the data in Table 2.2on page 18.

gen_gram	Grammar generator for program in lr.c
lr.c	Generates the grammar file for the sample grammar that LR states were generated in Table 2.2on page 18.
makeit	Make file for generating the grammar and running the modified YACC
g1, g2, g3, .. g10	These are the grammar generated by the gen_gram program that were fed to modified YACC program.

E.3.2 Subdirectory: word_count. A simple set of programs that reads in a dictionary and generates data for tables Table 2.11 and Table 2.11on page 31. These programs were also used to feed the data to the xgraph [1] program to generate the diagrams shown in Figure 2.5 and Figure 2.5, “Circsim-Tutor Lexicon Word Length,” on page 33.

E.3.3 Subdirectory: Version Two Performance Test. To test the lexicon access time in Circsim-Tutor version two, this directory was set up. It loops into all lexical entries and calls the access function and measure the time.

[1] An X-Windows utility on Sun machine that takes coordinates of a set of points and draws corresponding graph.

TEXTOUT.LSP	Supporting functions from version 2
finas.lsp	Lexicon access function from version 2
fixes	Simple functions to be able to run the files.
lexicon.lsp	Version2 lexicon
readme	Description of files and performance test.
symbols_dump.lsp	Test functions added to set the system up and access the lexicon.

E.4 File Length

To give an idea about the size of the program, in this section the line numbers of the C and Lisp files are presented.

Table E.1 File Length of Lisp Version Directories

Directory	Number of Lines
prs	18056
lexicon	4280
lexicon/autoload	1450
gram/lisp	1276
ali	3628
autoload	2247
TOTAL	30, 937

Table E.2 File Length of C Version Directories

Directort	Number of Lines
ali	3620
driver	413
gen_test	858
Lexicon	5711
prs:	9966
TOTAL	20, 568

APPENDIX F

CORRECTIONS of Original Thesis

F.1 Error In Implementation

On section 2.2.7.2 Page 31 of the thesis, there is a problem. The program that I wrote to generate the grammar [1]:

$$S \rightarrow X_i\, Z_i \qquad 1 \leq i \leq n$$
$$X_i \rightarrow Y_j\, X_i \mid Y_j \qquad 1 \leq i, j \leq n,\ i \neq j$$

had a bug. Instead of generating $X_i \rightarrow Y_j$, it generates: $X_i \rightarrow Y_i$, so the generated grammar is:

$$S \rightarrow X_i\, Z_i \qquad 1 \leq i \leq n$$
$$X_i \rightarrow Y_j\, X_i \mid Y_i \qquad 1 \leq i, j \leq n,\ i \neq j$$

The result is more states because of $X_i \rightarrow Y_i$. So by mistake, I actually created a worse grammar for LR parsers than the one that Fischer had in mind.

Although the original grammar has more number of rules (instead of a single $X_i \rightarrow Y_i$ for each i, in $X_i \rightarrow Y_j$ there are n-1 rules.) but the single rule $X_i \rightarrow Y_i$ generates a new state whereas in $X_i \rightarrow Y_j$, the state is shared with the rule: $X_i \rightarrow Y_j\, X_i$

F.1.1 Example

Here we give an example for n=5 of the generated grammars.

F.1.1.1 Original Grammar.

The original grammar (with the rule $X_i \rightarrow Y_j$) for n=5 is supposed to be:

$S : X_1\, Z_1$	$S : X_2\, Z_2$	$S : X_3\, Z_3$	$S : X_4\, Z_4$	$S : X_5\, Z_5$
$X_1 : Y_2$	$X_2 : Y_1$	$X_3 : Y_1$	$X_4 : Y_1$	$X_5 : Y_1$
$X_1 : Y_2\, X_1$	$X_2 : Y_1\, X_2$	$X_3 : Y_1\, X_3$	$X_4 : Y_1\, X_4$	$X_5 : Y_1\, X_5$
$X_1 : Y_3$	$X_2 : Y_3$	$X_3 : Y_2$	$X_4 : Y_2$	$X_5 : Y_2$
$X_1 : Y_3\, X_1$	$X_2 : Y_3\, X_2$	$X_3 : Y_2\, X_3$	$X_4 : Y_2\, X_4$	$X_5 : Y_2\, X_5$
$X_1 : Y_4$	$X_2 : Y_4$	$X_3 : Y_4$	$X_4 : Y_3$	$X_5 : Y_3$
$X_1 : Y_4\, X_1$	$X_2 : Y_4\, X_2$	$X_3 : Y_4\, X_3$	$X_4 : Y_3\, X_4$	$X_5 : Y_3\, X_5$
$X_1 : Y_5$	$X_2 : Y_5$	$X_3 : Y_5$	$X_4 : Y_5$	$X_5 : Y_4$
$X_1 : Y_5\, X_1$	$X_2 : Y_5\, X_2$	$X_3 : Y_5\, X_3$	$X_4 : Y_5\, X_4$	$X_5 : Y_4\, X_5$

This generates 46 grammar rules with 107 states.

[1] Fischer and LeBlanc [1988, p. 213]

F.1.1.2 Modified Grammar. The modified grammar (with the rule $X_i \rightarrow Y_i$) for n=5 is supposed to be:

$S : X_1 Z_1$	$S : X_2 Z_2$	$S : X_3 Z_3$	$S : X_4 Z_4$	$S : X_5 Z_5$
$X_1 : Y_1$	$X_2 : Y_2$	$X_3 : Y_3$	$X_4 : Y_4$	$X_5 : Y_5$
$X_1 : Y_2 X_1$	$X_2 : Y_1 X_2$	$X_3 : Y_1 X_3$	$X_4 : Y_1 X_4$	$X_5 : Y_1 X_5$
$X_1 : Y_3 X_1$	$X_2 : Y_3 X_2$	$X_3 : Y_2 X_3$	$X_4 : Y_2 X_4$	$X_5 : Y_2 X_5$
$X_1 : Y_4 X_1$	$X_2 : Y_4 X_2$	$X_3 : Y_4 X_3$	$X_4 : Y_3 X_4$	$X_5 : Y_3 X_5$
$X_1 : Y_5 X_1$	$X_2 : Y_5 X_2$	$X_3 : Y_5 X_3$	$X_4 : Y_5 X_4$	$X_5 : Y_4 X_5$

This generates 31 grammar rules with 187 states as reported in the thesis.

We tried the above grammar for n=100, which is by far out of reach of any LR based grammar (200 terminals, 102 nonterminals, 10,100 productions). The parser operated efficiently, even when we used fragments (when it has to consider all productions of the grammar). The choice set of each productions needs 200 bite=25 bytes. Therefore the size of the parse table is 25x10100=252,500 bytes, which is about 247KBytes.

We added all terminal symbols into the lexicon as they are: z_1 is of type ‘z_1’, and ‘y_1’ of type ‘y_1’ etc. The following inputs were tried:

Sentence	Time (seconds): Lisp	Time (seconds): C	Generated parse tree
y_1 z_1	0.32	0.016	(S (X_1 (y_1 'y_1'))(z_1: 1 'z_1'))
y_{100} z_{100}	0.33	0.016	(S (X_{100} (y_{100} 'y_{100} '))(z_{100}: 1 'z_{100}'))
z_1	0.17	0.001	No parse tree, type: z_1
z_{100}	0.19	0.001	No parse tree, type: z_{100}
y_1	0.35	0.001	(X_1 (y_1 'y_1'))
y_{100}	0.37	0.001	(X_{100} (y_{100} 'y_{100} '))

F.2 Indirect Left Recursion

A paragraph is added at the end of Section 3.2.4 on page 39.

F.3 LL(1) Table size

Page 27, formula 1 should be written as:

$$\mathrm{LLsize} = p \times \frac{t}{8} \tag{8}$$

Note that $t/8$ is the upper bound value, i.e. 7/8=1. This is mentioned in the text that every production has a set of terminal symbols (choice set), since one bit is enough to indicate present/absent of a terminal symbol in the set, we need $t/8$ bytes in each set.

This changes the table: 2.2 on page 32 to (only LL Size column is changed):

Table F.1 LR States for Fischer Grammar

n	No. of Terminals	No. of Nonterms	No. of Prod	No. of States	LR Size	LL Size
1	4	2	3	5	35	3
2	6	3	7	14	140	7
3	8	4	13	35	455	13
4	10	5	21	82	455	42
5	12	6	31	187	1,558	62
6	14	7	43	422	13,284	86
7	16	8	57	947	23,675	114
8	18	9	73	2,114	59,192	219
9	20	10	91	4,691	145,421	273
10	22	11	111	10,342	351,628	333
11	24	12	133	22,651	838,087	399

Table F.1 shows the numbers, in LR states, the marked numbers with '*' display the states that need two bytse (> 256)

Table F.2 LR States for Fischer Grammar

n	No. of Terminals	No. of Nonterms	No. of Prod	No. of States	LR Size	LL Size
1	2	2	3	4	30	3
2	4	3	6	13	104	6
3	6	4	12	34	474	12
4	8	5	20	81	1,134	40
5	10	6	30	186	3,162	60
6	12	7	42	421*	16,840	84
7	14	8	56	946*	43,516	112
8	16	9	72	2,113*	109,876	216
9	18	10	90	4,690*	272,020	270

Table F.2 LR States for Fischer Grammar

n	No. of Terminals	No. of Nonterms	No. of Prod	No. of States	LR Size	LL Size
10	20	11	110	10,341*	661,824	330
11	22	12	132	22,650 *	1,585,500	396

F.4 Figure reference

On page 89, after figure 3.5, the reference is to Figure 3.6.

On page 157, instead of figure 5.30 the referce is on Figure 19.

F.5 Changed Table

The table 5.8 on page 171 of thesis, is changed to the following table. The main part is the winner in third entry is no one. This was a typing error. In the Ph.D. proposal, page 79, table 18, and also in Ph.D defense exam presentation, the right value (Equal) was reported. Corrected table is:

Table F.3 Comparison of 3 Way Match with 4 Way Match

current error	next type of error	3 way comp. number of errors	4 way comp. number of errors	Algorithm which reports fewest extraneous errors
Missing character	**No error**	1	1 or 2	3 way?
	Reversed order	2	1	4 way
	Missing character	2	2	EQUAL
	Added character	2	1	4way
Added character	**No error**	1	1 or 2	3 way?
	Reversed order	2	1	4 way
	Missing character	2	1	4 way
	Added character	2	2	EQUAL

F.6 Indirect Left Recursion Handling

A new section for handling indirect left recursion is added to the parser chapter. Section 3.2.4 on page 39 of this document is modified. Two subsections: 3.2.4.1 3.2.4.2 are added.

F.7 Chart Table

The final chart for the parser example at the end of Section 3.4 , right after the Diagram M, the fisrt entry S(10, 2) in the first entry (input #0) is changed to S(10, 1). Note that there is only a single S $\longrightarrow$ NP (01)(VP 1 10)(which is the second production. But there are two ways to use the first production: S $\longrightarrow$ S PP. The first one is: S $\longrightarrow$ S(0 4)PP(4 10) and the second one is: S $\longrightarrow$ S(0 7)PP(7 10).

F.8 Word Table

Section 6.6 Word Table, the last sentence in the first paragraph is deleted. The item was description of the lexicon table which wrongly was copied in word table description.

F.9 Word Segmentation

In addition to word numbering difficulties, and need for future reseach, consider the following problem with the input: ... the cons equence is ...

Note in version 12 of the lexicon, we have the following words:

- co abbreviation of cardiac output
- ns abbreviation of nervous system
- sequence as single word

The spelling corrector is called with ‘cons’. Since ‘co’ is before ‘consequence’ in the word table (alphabetically sorted), the spelling corrector compare cons with co, and it concludes that it can split ‘cons’ into two valid words: co, and ns.

Since we terminate the spelling process after detection of a word boundary error, it does not even compares ‘cons’ with ‘consequence’.

The solution, as is described in the future research in Section 9.2.3 on page 174, is to allow multiple replacement candidates for all three cases:

- single words
- Word boundary error
- phrases in the system.

Currently for the first case we have 10 replacement candidates and for the latter two cases a single replacement. The difficulty is word numbering becomes a complicated issue if one replacement is a single word and another one is a join word with teh next input word, ...

APPENDIX G

10: Additions

10.1 Parser Algorithm.

The parser is an extended LL parser. Instead of the traditional top down parsing algorithms that the parser starts from the first word and go forward until it reaches the end of the input, the algorithm actually parses segments of the input at a time and combine them as it progresses. The parser departs from the LL(1), in handling the nonterminal symbols on top of the stack. The underlying parser, parses one production at a time.

```
parse(nonterminal N, inputLocation loc)
    Symbols = inputAtLocation(loc)
    FOR each part of speech (pos) of Symbol DO
        FOR each production (p) of N DO
            IF pos IN choiceSet(p)
                parse_prod(p, loc, pos)
            end if
        end for
    end for
end parse
```

For each nonterminal entry in the chart the following parameters are recorded:

1. The first word's part of speech (left most leaf node in the parse tree).
2. The location of the first input word. This is the location that the nonterminal enters the chart.
3. The location of its last input word, right most leaf node in its parse tree, plus one -so we can manage the null drivers symbols.
4. The production number.

In production-parsing the production right hand side is loaded in a newly created parse stack. Associated with the stack is the start location of input (loc) and the production number and the input word part of speech *pos*. For the production number p: $N \rightarrow s_1\ s_2 .... s_n$ the stack is loaded as: (s_1 loc) (s_2 ?)... (s_n ?). At the end of the parsing of the production, the stack looks like:

(s_1 loc i pos) (s_2 i j pos_1)... (s_n j k pos_h)

N is entered in the chart location *loc* as *N(k p pos)* along with the parsed production: (s_1 loc i pos) (s_2 i j pos_1)... (s_n j k pos_h). This says that the nonterminal N, using its production number p, has parsed the input segments from location *loc*, with part of speech *pos*, up to (but not including) location k. Here is the details of production-parsing:

The parser moves from left to right, and processes the top of stack symbol (s_m i), where n $\geq m \geq 1$, and the next input word part of speech *pos*:

1. When s_m is a terminal symbol:
   1.1. If s_m is not the same as *pos*, reject parsing of this branch. If they match, then
   1.2. replaces the next symbol of stack: (s_{m+1} ?) with (s_{m+1} i+1).
   1.3. reads in the next input word, and for each part of speech of the next input word the stack is replicated (for efficiency purposes, the replication is delayed until we know for sure a part of speech is acceptable).
2. When s_m is an action symbol: its associated code[1] is executed. Also the parser replaces the next symbol of stack: (s_{m+1} ?) with (s_{m+1} i).
3. When s_m is a nonterminal symbol:
   3.1. The parser searches the chart location *i* for the existence of the parse tree of s_m:
      - If the search fails, it calls parse (s_m, i).
      - If s_m is suspended at location *i* of the chart, suspend the production.
   3.2. Replaces the next symbol of stack: (s_{m+1} ?) with (s_{m+1} j).
   3.3. If there are more than one parse tree for s_m (there are multiple entries of: s_m(j_1 p_a pos), s_m (j_2 p_b pos),... s_m(j_n p_c pos)), for each distinct value of j (in j_1, j_2,..., J_n) replicates the stack, and replace (s_m i) with (s_m i j pos). Two points are worth noting here:
      - the production number p_n, in s_m(j_i p_n pos), is ignored.
      - only distinct values of j are used.

   The above two points show that regardless of how many ways the nonterminal s_m has parsed input segment <i, j>, a single entry is saved in the current production of N that is using s_m. This is where the efficiency of the parser comes to surface: the ambiguity is packed right at its origination point, and is not propagated to the higher parse trees.
4. Move the stack pointer to the right so the next symbol (s_{m+1} j) will be on the top of stack.

10.2 Ambiguity Handling.

In an ambiguous environment such as natural language, the parser must be able to slow down the spread of ambiguity. Otherwise, the rapid growth of ambiguity brings the parser to its knees. The usage of the chart has helped to confine the spread of ambiguity. Basically, it has eliminated multiplication factors of spread.

When there is more than one parse tree of N that have common leaf nodes, higher level trees treat all subtrees as a single node. So, if there are m entries of N in the chart, but only there are n distinct leaf nodes (n $\leq m$), higher level trees treat N as having n parse trees. Parse trees of N are sorted in ascending order of input consumption. We only care how many distinct segments of inputs are parsed by the nonterminal s, not how many ways each segments are parsed.

When we have the nonterminal s_m on top of the parse stack with the input

[1]Our parser table generator assigns an internal (action) code for augmented functions in the grammar. The parser will access associated functions by using the action code.

word at location *loc*, and part of speech *pos*, we search the chart location *loc* for entries: s_m(k **?** pos), ignoring production number). Then, for each distinct value of k, we replicate the parse stack. So by ignoring the production numbers, and simply considering one of many parse trees with the same leaf nodes, we put a break on spread of the ambiguity.

To clarify the concept, lets try to parse the nonterminal S using the production: S $\rightarrow$NP VP. The algorithm calls parse (NP, 0). Lets assume that *n* parse trees are found for NP, where n>0 (for n=0, the parsing simply fails). For each distinct segment of the input that is parsed as NP, the system continues parsing for VP. In other words, if there are n_f *(where* $n_f \geq 1$*)* parse trees of NP that contains the segment (0, i_f) of input, we call *parse (VP,* i_f*)* only one time. If the parsing for VP starting at location i_f succeeds then, for each distinct segment (i_f, j_f) of input, a single entry S(j_f *prod pos1*) with production: S $\rightarrow$(NP 0 i_f pos_1)(VP i_f j_f pos_2) is entered in the chart at location 0.

In Table 10.1, the columns "No. of Trees" provide number of parse trees for nonterminals NP, VP, and S. The columns "Segment" provide the parsed input segments for each nonterminal.

Table 10.1

NP		VP		S	
No. of Trees	Segment	No. of Trees	Segment	No. of Trees	Segment
n_1	$0\text{-}i_1$	m_1	$i_1\text{-}k_1$	$n_1 \times m_1$	$0\text{-}k_1$
n_2	$0\text{-}i_2$	m_2	$i_2\text{-}k_2$	$n_2 \times m_2$	$0\text{-}k_2$
n_3	$0\text{-}i_3$	m_3	$i_3\text{-}k_3$	$n_3 \times m_3$	$0\text{-}k_3$
...	...	...	...	...	...
n_h	$0\text{-}i_h$	m_h	$i_h\text{-}k_h$	$n_h \times m_h$	$0\text{-}k_h$

The given number of parsed trees for S is the actual number that any conventional parser would produce, which is: ($n_1 \times m_1 + n_2 \times m_2 + n_3 \times m_3 + \ldots + n_h \times m_h$). With our technique, for each segment 0-k_f, there is one entry in the chart. Therefore, we will have only *h* new entries in the chart, for parsing the production S $\rightarrow$NP VP, after NP, and VP are parsed.

The above technique controls the combinatorics of the parse trees. This is the main strength of our parser and is a feature that makes it a better choice for ambiguous environment such as natural language. An important difference with Tomita's parser is that in his parser ambiguity is spread, and later the parser packs the local ambiguity, but ours does not allow the ambiguity to spread. In Tomita's parser, the program examines the graph-structured stack to do *local ambiguity packing* . This requires the parsing trees to be constructed and then the system eliminates the duplicates. For sentences with a few parsing trees, this method works fine. But, in a sentence with a lot of ambiguity the ambiguity grows so fast and crashes the system before the parsing trees are built. An important issue is to remember that not all the time all productions succeed parsing. For instance, in a production: N $\rightarrow s_1\ s_2\ ....\ s_m\ ....\ s_n$, if the parsing fails at s_m, what is the use of building all possible parse trees up to getting to s_m and failing? In any event, if we are going to pack the trees, why should we waste time to have them expanded? Note that we achieved ambiguity packing without much effort. Also, we are not losing any information and there is no need for unpacking.

Note that generating the full parse trees from the entries in the chart is a very straight forward, and fast task. The data is readily available.

10.3 Example.

The following example is provided to give a trace of the parser algorithm. The grammar and the lexicon (simple and limited) are borrowed [Tomita, 1986]. The grammar is shown in the table below. The ‘choice set’ column shows the expected input entry to expand the corresponding production. The ‘conflict symbol’ column shows the input words that cause the conflict between the associated production and the production number(s) given in the ‘conflict with prod. no.’ column.

prod.	production	choice set	conflict symbol	conflict with prod. no.
1	S → S PP	noun det	noun det	2
2	S → NP VP	noun det	noun det	1
3	NP → noun	noun	noun	5
4	NP → det noun	det	det	5
5	NP → NP PP	noun det	noun det	3, 4
6	PP → prep NP	prep		
7	VP → verb NP	verb		

A very small lexicon of nine words (where each word has a single sense, with no regard to lexical ambiguity) was considered for this example. The emphasis is on syntax.The words and their parts of speech are:

Word String	Part of speech
I, man, park, telescope	noun
a, the	determinant
in, with	preposition
saw	verb

We parse the infamous sentence: *I saw a man in the park with a telescope*. We consider that ‘*I*’ is at location zero of input, ‘*saw*’ at location 1 and ‘*telescope*’ at location 9.

As mentioned in Section 10.1 when the production number p: N $\rightarrow s_1\ s_2$ finishes parsing input segment <begin end>, N is entered in the chart location *begin* as *N(end p pos)* along with the parsed production: (s_1 begin i pos)(s_2 i end pos_1). For simplicity, we drop part of speech here, since each input word has a single part of speech. Also, because of the limited number of productions the production numbers are dropped. The chart entry for N will be shown as: $N_{end} \rightarrow (s_1\ begin\ i)(s_2\ i\ end)$.

The parsing starts with *parse(s 0)*. Productions 1 and 2 of S are valid LL(1) choices for this nonterminal. Productions 1 is suspended because of recursive symbol

S. Parsing production (2): $S \rightarrow (NP\ 0)(VP\ ?)$, causes parsing NP at location 0. The LL(1) choice sets of *NP* allow expansion of the third and the fifth productions for 'noun':

(3) NP→ (noun 0) (5) NP → (NP 0)(PP ?)

The symbol 'noun' in (3) matches the input 'I', and NP enters the chart as $NP_1 \rightarrow (noun\ 0)$. In (5) we have a recursive nonterminal NP, but NP_1 is already in the chart, therefore parsing resumes for its right symbol PP, at location 1. PP fails, since the input symbol at location 1 is a verb, and is not in PP's choice set.

We get back to production (2). Since NP_1 entered the chart, VP, its right sibling , is activated and parsing starts at location 1. After the expansion of VP using production 7, the terminal symbol 'verb' matches the input 'saw' and we get to: (7) VP → (verb 1)(NP 2). NP resumes parsing at input location 2. Productions 4 and 5 of NP accept input symbol 'a':

(4) NP→ (det 2)(noun ?) (5) NP → (NP 2)(PP ?)

Production 4 has two terminal symbols that match the next two input symbols. Production 4 terminates and causes insertion of NP into the chart at location 2 as NP_4→ (det 2)(noun 3). In (5) we have a recursive nonterminal NP, but NP is already in the chart as (NP 2 4), therefore parsing resumes for its right symbol PP, at location 4: (6) PP → (prep 4)(NP ?)

Input word *in* at location 4, matches *prep*, and we get to: (6) PP → (prep 4)(NP 5). We have input word *the* , with part of speech: *det*, at location 5. Parsing NP at location 5:

(4) NP→ (det 5)(noun ?) (5) NP → (NP 5)(PP ?)

Production (4) terminates and NP enters the chart location 5 as: NP_7 → (det 5)(noun 6). Production (5), now that NP enterd the chart, gets activated as: NP → (NP 5 7)(PP 7) and parsing resumes for PP at location 7. This causes production parsing of: (6) PP → (prep 7)(NP ?).

Input word *with* , at location 7, matches *prep*, and the parsing continues at location 8 for NP. We have input word *a*, with part of speech: *det*, at location 8. Parsing NP at location 8:

(4) NP→ (det 8)(noun ?) (5) NP → (NP 8)(PP ?)

Production (4) terminates and NP enters the chart location 8 as: NP_{10} → (det 8)(noun 9), and p roduction (5) gets activated as: NP → (NP 8 10)(PP 10). Activating PP at location 10 fails since PP only accepts inputs of type *prep*, but we have reached the end of the input. So parsing of production (5): NP → (NP 8)(PP ?) fails.

Getting back to production: (6) PP → (prep 7)(NP 8), after NP's parsing terminated, PP enters the chart alocation 7, as: PP_{10} → (prep 7)(NP 8 10). Then production-parsing of production of production 5 terminates, and NP enters the chart location 5, as: NP_{10} → (NP5 7)(PP 7 10). No further expansion of the recursive production 5 is possible because PP has failed at location 10 of the input.

In for production 7: VP → (verb 1)(NP 2), there are four entries of NP at location 2 of the chart:

NP_4 → (det 2)(noun 3) NP_7 → (NP 2 4)(PP 4 7) NP_{10} → (NP 2 4)(PP 4 10)
NP_{10} → (NP 2 7)(PP 7 10)

VP enters the chart as:

$VP_4 \rightarrow$ (verb 1)(NP 2 4) $VP_7 \rightarrow$ (verb 1)(NP 2 7) $VP_{10} \rightarrow$ (verb 1)(NP 2 10)

Note that there are two ways to parse (NP 2 10), and therefore, two ways to parse (VP 1 10), but there is a single entry for VP at location 1 of the chart, $VP_{10} \rightarrow$ (verb 1)(NP 2 10). Although VP is isolated from the multiple parse trees of NP, it can access them both.

After VP enters the chart, parsing production (2) of S is terminated and S enters the chart as:

$S_4 \rightarrow$ (NP 0 1)(VP 1 4) $S_7 \rightarrow$ (NP 0 1)(VP 1 7) $S_{10} \rightarrow$ (NP 0 1)(VP 1 10)

Now we resume parsing of the suspended production: (1) S $\rightarrow$ (S 0)(PP ?). There three entries: S_4, S_7 and S_{10} in chart location 0. For S_4 we get to: S $\rightarrow$ (S 0 4)(PP 4). There are two entries for PP at location 4:

$PP_7 \rightarrow$ (prep 4)(NP 5 7) $PP_{10} \rightarrow$ (prep 4)(NP 5 10)

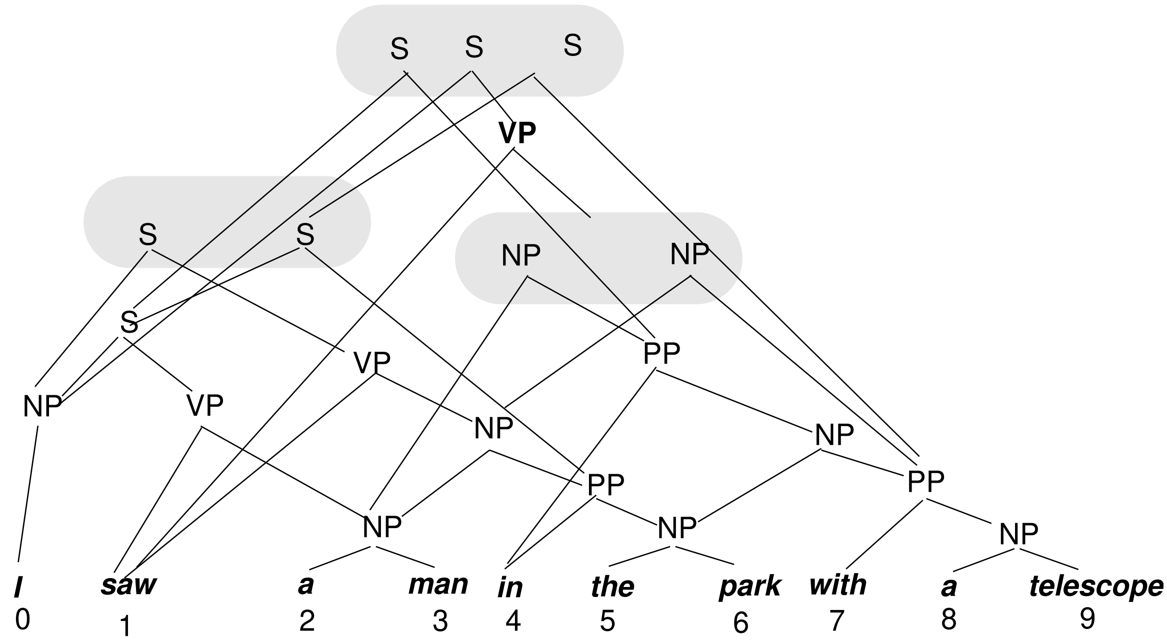
This will cause the insertion of S at location 0 of the chart as:

$S_7 \rightarrow$ (S 0 4)(PP 4 7) $S_{10} \rightarrow$ (S 0 4)(PP 4 10)

For S_7 we get to: S $\rightarrow$ (S 0 7)(PP 7). Although there are two entries for S_7, only one will be considered, and $S_{10} \rightarrow$ (S 0 7)(PP 7 10) will be inserted into the chart. For S_{10} we get to: S $\rightarrow$ (S 0 10)(PP 10). But, PP failed parsing at location 10 of the input, therefore, the parsing of S $\rightarrow$ (S 0 10)(PP 10) fails. The final chart is:

Input #	Symbol	Recognized Nonterminal/Production		
0	I	$NP_1 \rightarrow$ (noun 0) $S_4 \rightarrow$ (NP 0 1)(VP 1 4)	$S_7 \rightarrow$ (S 0 4)(PP 4 7) $S_7 \rightarrow$ (NP 0 1)(VP 1 7)	$S_{10} \rightarrow$ (S 0 4)(PP 4 10) $S_{10} \rightarrow$ (S 0 7)(PP 7 10) $S_{10} \rightarrow$ (NP 0 1)(VP 1 10)
1	saw	$VP_4 \rightarrow$ (verb 1)(NP 2 4)	$VP_7 \rightarrow$ (verb 1)(NP 2 7)	$VP_{10} \rightarrow$ (verb 1)(NP 2 10)
2	a	$NP_4 \rightarrow$ (det 2)(noun 3)	$NP_7 \rightarrow$ (NP 2 4)(PP 4 7)	$NP_{10} \rightarrow$ (NP 2 4)(PP 4 10) $NP_{10} \rightarrow$ (NP 2 7)(PP 7 10)
3	man			
4	in		$PP_7 \rightarrow$ (prep 4)(NP 5 7)	$PP_{10} \rightarrow$ (prep 4)(NP 5 10)
5	the		$NP_7 \rightarrow$ (det 5)(noun 6)	$NP_{10} \rightarrow$ (NP5 7)(PP 7 10
6	park			
7	with			$PP_{10} \rightarrow$ (prep 7)(NP 8 10)
8	a			$NP_{10} \rightarrow$ (det 8)(noun 9)
9	telescope			

The parse trees are:



10.4 Adaptive Disagreement Threshold.

To reduce the time spent on comparing the erroneous word with irrelevant words from the lexicon, we put a limit on the number of mismatches and use a percentage of the length of the misspelled word. For example, “specification” and “sp**ice**fication” differ by two characters out of thirteen, so there is a 16% (2/13) disagreement. The words ‘the’ and ‘tae’ differ by one character out of three, so they have a 33% (1/3) disagreement. The disagreement threshold is used to terminate the comparison of an irrelevant word with the erroneous word, in effect acting as a filter. If the number is too high (a loose filter), we get many irrelevant words. If the number is too low (a tight filter), a lot of good candidates are discarded. For this reason, we came up with an adaptive method that dynamically lowers the tolerance for errors as better replacement candidates are found.

In order to have a wide acceptance capacity, we start with a limit of 50% disagreement. As the first candidate is found with the value of disagreement n, the limit is dynamically set to the maximum of 30% and n. Both upper bound and lower bound disagreement (50% and 30% mentioned here), are configurable.The whole idea is to have a system that, in the worst case, can find a replacement candidate, and as the candidates are found, it can dynamically look only for better candidates. After the first ten candidates are found (the number is configurable), the limit is set to the maximum value of disagreement amongst the candidates found so far. At any stage, if the disagreement is less than 10%, the process terminates and the comparison word will be considered the only replacement for the misspelled word. Note that the upper limit of 50% allows our algorithm to correct a misspelled word of length two. Any limit lower than 50% does not allow correction of a single letter in a two character word, unless there is specific algorithm that applies to small size words.

A graphical representation of this algorithm is presented in Figure 1. The disagreement between the misspelled word and the word from the lexicon is shown in form of a ball. The bigger the disagreement, the larger the size of the ball. The ball is rolling down on a surface with three traps. As the ball rolls down it falls into the first trap that is larger than its size. Each one of the traps has a trigger. When the ball falls in a trap, the associated trigger is activated, and the ball is collected in the associated container. Once the container is full, it drops the biggest size ball (least likely replacement word), activating the fourth trigger. The four triggers perform the following actions:

1. Trigger one terminates the comparison process. This word is an acceptable candidate for replacing the string containing the error.
2. Trigger two closes the extended disagreement trap. Once we find a word that meets or betters the middle disagreement, there is no need for extended disagreement.
3. Trigger three lowers extended disagreement to the disagreement of this word. We have found a word that has a disagreement less than the extended one, we accept no match worse than this.
4. Trigger four lowers average disagreement to the worst one in the container, and closes the extended disagreement. This action forces us to look only for words that are better than the worst match that we have found so far.

10.5 Cooperative Processes for Spelling Correction.

In Our system the interleaved processes: lexical analysis, spelling correction, syntax analysis, and semantic analysis work together to find the best matching word from the lexicon for a misspelled word.

The steps for spelling correction and the choice of the best candidates are organized as follows:

1. Detection: The lexical analyzer detects that the next input word is misspelled.
2. Correction: The spelling corrector creates a list of one or more replacement words for the misspelled word: ($(w_1\ e_1)... (w_n\ e_n))$. Where, w_i is the replacement word from the lexicon, and e_i is the associated error weight. The list is sorted in ascending order of e_i.
3. Reduction: The phrase recognizer checks whether any word in the list can be combined with the previous/next input word(s) to form a phrase. If a phrase can be constructed, the word that is used in the phrase is considered the only replacement candidate for the erroneous word.
4. Part of speech assignment: For each part of speech p of w_i, a sub-list $(p_i\ w_i\ e_i)$ is created. So now the list is: ($(p_{11}\ w_1\ e_1)\ (p_{12}\ w_1\ e_1)... (p_{nm}\ w_n\ e_n))$.
   - 4.1. The list is sorted by parts of speech. If w_i and w_j have a common part of speech, p_n, then a sub-list of replacement words: $(p_n\ (w_i\ e_i)\ (w_j\ e_j)...)$ is created .
   - 4.2. The sub-lists are put in the replacement list in ascending order of the first word's error weight in each sub-list: $((p_1\ (w_i\ e_i)\ (w_j\ e_j)...)\ (p_2\ (w_k$

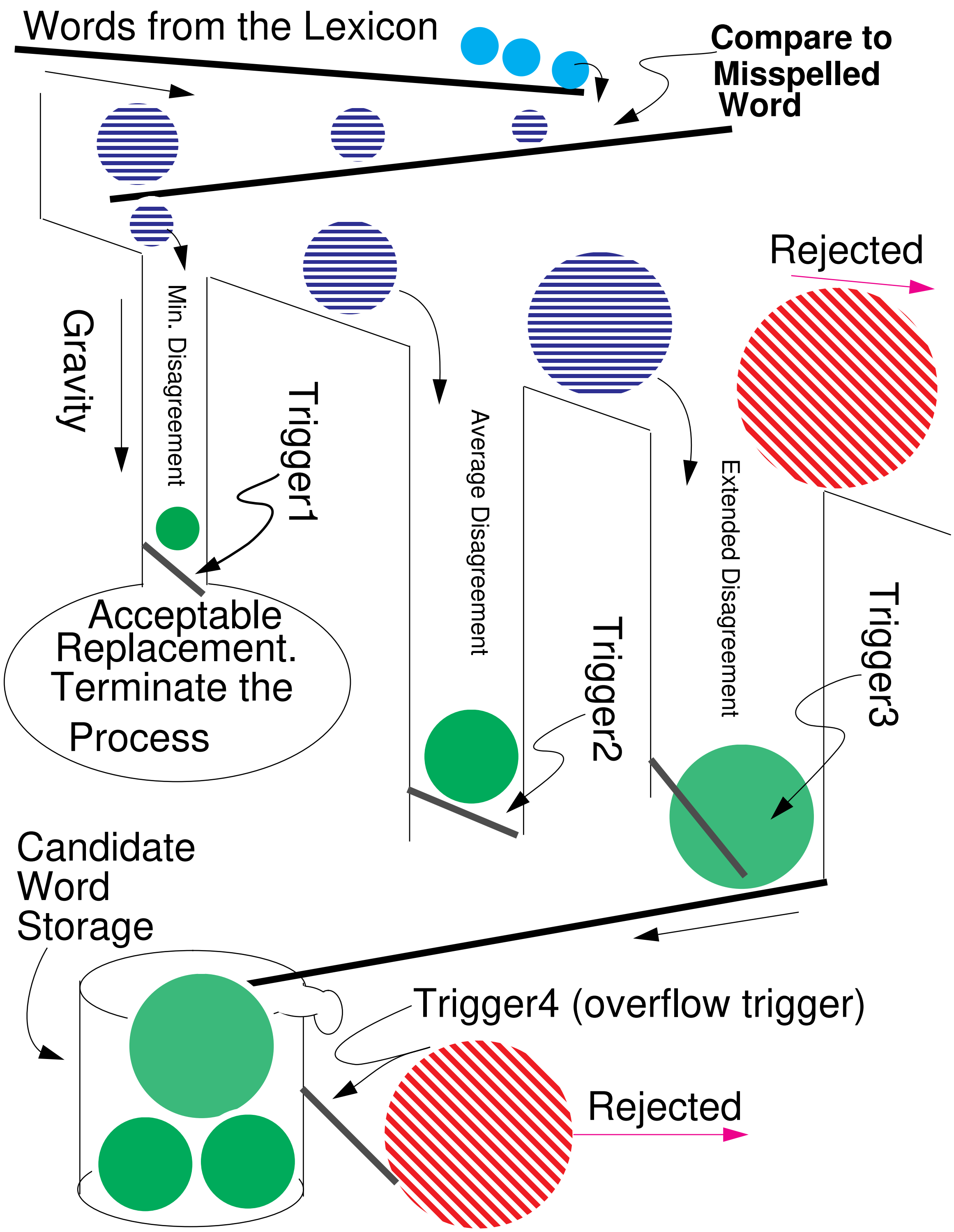


Figure 1. Graphical Representation of Comparison Algorithm

$e_k)$ $(w_m\ e_m)...)...)$. In this list, $e_i \le e_j$, $e_i \le e_k$, *and* $e_k \le e_m$. There is no enforced relation between e_j and e_k (or e_m).

5. Syntax analysis: The error weights are dropped from the list, and the final replacement list: $((p_1\ w_i\ w_j\ ...)\ (p_2\ w_k\ w_m\ ...)...)$ is passed to the parser. The

parser examines each sub-list and discards the sub-lists that violate the syntactic rules. Obviously, the syntax analysis process cannot distinguish among the words with the same part of speech. Thus, the parser either accepts or rejects all words of a sub-list.

6. Semantics analysis: Decisions among the candidates with the same part of speech is made by semantics rules. LFG annotations from the lexicon and the grammar are used to build a functional representation of the input sentence. Many of the candidate words, even though they may have the same part of speech are rejected when they violate functional well-formedness constraints.

Now we describe the cooperation amongst the processes (lexical analyzer, phrase recognition process, syntax analyzer and semantic analyzer) for spelling correction by means of examples covering different aspects of the method.

10.5.1 Join/Split words. The distinction for missing/extra characters at the end of a misspelled word allows the correction of sentences like: '*I saw a man witha tele scope*'. The program splits the string: 'witha' in two words: 'with a', and joins the two strings: 'tele' and 'scope' to a single word: telescope.

10.5.2 Use of Phrase Recognition. Suppose the user types a sentence containing the following two words: "*card contr*". The lexical analyzer concludes that 'card' is a misspelled word, since we do not have the word 'card' in our lexicon. It calls the spelling error process. The spelling corrector returns a list of replacement words in order of error weight: *cardiac, carbon, cardiovascular, cancer.*

The lexical analyzer passes this list to the phrase recognition process, since the first word in the list can be part of a phrase. The phrase recognizer, in turn, calls the lexical analyzer to fetch the next input word. The lexical analyzer, fetches the word 'contr', and because it cannot locate it in the lexicon, calls the spelling corrector. The spelling corrector returns a list of replacement words in order of error weight:

control, contract, contracts, controlled, controller, contractile, contraction, contractility, confuse

The lexical analyzer passes all the replacement words to the phrase recognizer. The phrase recognizer concludes that we have the phrase: "cardiac contractility".

Note that the word "*cardiac*" is the first choice for the misspelled (actually abbreviated) word 'card' and the word "*contractility"* is one to the last choice for the misspelled (actually abbreviated) word 'contr'. However, since they make up a phrase in our lexicon, the lexical analyzer simply returns a single choice *"cardiac contractility"* to the parser.

10.5.3 Use of the Parser. Suppose the user types the following sentence: *lohng sees Mary*. The lexical analyzer calls the spelling error to correct the misspelled word: *lohng*. The spelling corrector returns a list of words, in order of the error weights: *long, longer, linger, John, length*. The lexical analyzer assigns part of speech:

((**noun**: *long, John, length*)(**adj**: *long, longer*)(**adv**: *long*)(**verb**: *long, linger*))

The parser, in turn, only accepts words with part of speech ‘noun’, as the other parts of speech are not syntactically correct. Therefore the list is reduced to three words: *long, John, length*. Note that the parser cannot distinguish among these three words, as they all are of type ‘noun’.

10.5.4 Use of the Semantic Information. The parser passes the list (*long, John, length*) to the LFG functional-structure construction process. The second input word ‘*sees*’ requires an animate *subject*. The word *John* is the only word that can be the subject for the verb *sees*. Therefore, the functional-structure builder process eliminates the words *long* and *length* from consideration. Thus the system concludes that the only replacement word for *lohng* is *John*, and the sentence becomes: *John sees mary*.

11. Empirical Results

In this section, we present two empirical test results of the parser performance. First, we show the parser behavior in very ambiguous, long sentences. Second, we show the parser toleration for a huge grammar. In both cases, a comparison is made with Tomita’s parser.

11.1 Ambiguity Handling.

To generate really complicated strings of words that are highly ambiguous, using the grammar:

S → S PP	S → NP VP	VP → verb NP	PP → prep NP
NP → noun	NP → det noun	NP → NP PP	

the sample sentences are created from the following schema [Tomita, 1986]:

$$\textit{noun verb det noun (prep det noun)}^{n-1}$$

The fact that the ambiguity multiplies makes the above sentence very difficult to parse for large values of n. Every increment of n causes more difficulty. As a result, the whole forest of the parse trees is affected. The snowball effect of ambiguity requires an efficient parser to deal with a large volume of parse trees. Our parser, by using the chart, stores the tree linearly and that is the trick of handling this huge ambiguity even for large values of n.

The Table 11.1 gives a summary of parse statistics for the above sentence for n=1 to n=40. This is the result of running the test on a Sun Sparc 5, using the *Carnegie Mellon University Common Lisp*. Tomita’s parser crashed at n=22 (after a few hours). Although the number of ambiguous readings is extremely high, the parser operates efficiently in both time and space. The headers in the table are[1]:

- N is the number of the propositions (as in the formula).
- Time is the CPU time to parse the sentence in *milliseconds* for both parsers.
- Entries in the chart is the number of productions that are actually saved in the chart.

[1]We used Tomita's “The Generalized LR Parser/Compiler Version 8.1" toolkit for comparison with our parser. Note that we had to put some LFG rules in the grammar with Tomita’s parser, since it would not run otherwise. Although we put very simple LFG rules, x0 = x1 (which is equivalent of ↑=↓) to every production, it still lowered the number of valid parsed trees. For instance, at n=16, we have over 35 million ambiguities which is bigger that 11,048,576 ambiguities that Tomita’s parser reported for n=21.

- Actual Ambiguity: is the number of parse trees for the given sentence.

Table 11.1 Result for n=1 to n=50

n	CPU Time in millisecond		Entries in the chart	Actual Ambiguity
	Tomita's	Ours		
2	140	0	10	2
4	170	10	35	14
6	270	20	84	132
8	450	40	165	1,430
10	700	90	286	16,796
12	1,110	80	455	208,012
13	1,420	90	560	742,900
14	1,860	120	680	2,674,440
16	3,810	180	969	35,357,670
18	29,640	300	1,330	477,638,700
20	361,410	400	1,771	6,564,120,420
21	2,469,780	480	2,024	24,466,267,020
22	crashed	590	2,300	91,482,563,640
25		940	3,276	4,861,946,401,452
30		2,250	5,456	3,814,986,502,092,304
35	crashed	4,380	8,436	3,116,285,494,907,301,262
40		10,660	12,341	2,622,127,042,276,492,108,820

As an example, for n=7 we can make up a sentence like:

> *I* saw a man, on the bed in the bedroom, in the apartment, on the main street, at night, in Chicago.

This sentence has 429 parse trees. However, only 120 entries are saved in the chart.

The main reason for a huge difference of speed, and tolerance for much larger ambiguity, between the two parsers is the fact that in our parser, the ambiguity is not propagated, and the system does not allow it to spread. In Tomita's parser, first the parse tree is built, and then it is packed. The number of ambiguity for large value of n is so high, that it is computationally next to impossible to allow the full construction of trees to proceed.

11.2 Dealing with a Big Grammar.

The parser is able to handle very large grammars that no bottom up based parser can tackle. In the worst case, the number of states in an LR parser is exponential in the size of the grammar. Fischer and LeBlanc [1988, p. 213], give the following grammar that has $O(n^2)$ productions, and has $O(2^n)$ LR states:

$$S \rightarrow X_i Z_i \qquad 1 \leq i \leq n$$
$$X_i \rightarrow Y_j X_i \mid Y_i \qquad 1 \leq i, j \leq n, i \neq j$$

We tried the above grammar with Tomita's system. For n=8, there are 2114 states. For n=9 it crashed. This is an inherent problem of LR parsers in general. In fact, we tried this grammar with the Unix famous YACC, a LALR(1) parser generator. YACC gives up at n=8 as the number of states grows beyond 1000. We took the YACC source code from the Sun machine, and modified it to raise the above limits. For n=10, it took over five minutes for the new program to generate 10,342 states. For n=11, it took over ten minutes for the new program to generate 22,651 states, before it crashed, while generating the action table.

We tried the above grammar for n=100, which is far out of reach of most LR based grammar (the number of states here will be: $O(2^{100})$ states). It has 200 terminals, 102 nonterminals, 10,100 productions. The generated parser table for this grammar is about 131 Kilobytes. With this huge grammar, the parser operated efficiently, even when we used fragments (when it had to consider all productions of the grammar).

The parser is capable of handling productions like: $\mathbf{S} \rightarrow \mathbf{S}\ \mathbf{S}$. Adding this production to the sample grammar of Section 11.1 and the following sentence: "*I found the coin, you lost a coin*" generates the following parse tree:

(S (S (NP (noun ***I***))
(VP (verb ***found***) (NP (det ***the***)(noun ***coin***))))
(S (NP (noun **you**))
(VP (verb ***lost***) (NP (det ***a***)(noun ***coin***)))))

12. Conclusion

The main strength of this parser is its ability to handle all natural language parsing requirements, as specified above. It uses a minimal amount of storage. It has also preserved the LL time efficiency and its rich structure to be used later for error recovery.

The parser meets the challenges facing a NLP in an integrated environment. Namely, it:

1. demonstrates speed and space efficiency.
2. deals with ambiguous sentences.
3. enforces the semantic rules and allows augmentation of semantic rules in the grammar.
4. handles ill-formed input such as fragments and ellipses.
5. recognizes the phrases within a sentence.
6. accepts input words with multiple parts of speech and multiple senses.
7. uses syntactic/semantic information to help correct spelling errors.

In comparison with Tomita's parser:

1. Faster parser with smaller table size.
2. Better handling of ambiguity, by controlling the spread of ambiguity.
3. Better handling of fragments, no need for addition of productions for fragments.

We have improved spelling correction by combining parsing and spelling correction that makes more use of context than existing systems.

1. The spelling corrector is an extension of three way match algorithm to improve better comparison results.
2. Instead of a simple error distance calculation for comparing the two words, a position sensitive error weight has given us many advantageous:
    - 2.1. A better method for comparing different replacement candidates.
    - 2.2. Allowing us to consider adjacent keyboard character problem, and similar sound character errors for a keyboard entry system.
    - 2.3. Extendable to work with spelling errors produced in other types of data entry like Optical Characters Recognizer.
    - 2.4. Allowing us to effectively deal with word boundary problem and random abbreviation, a task that is absent from most other systems.
3. An adaptive disagreement limit has produced a filtering system for removing irrelevant words from the lexicon. In early stages of the comparison, the system starts with a loose filter, and tightens the filter as better replacement candidate words are found.
4. A segmented lexicon that allows the spelling corrector to be selective about searching the next segment of the lexicon instead of the usual sequential search.

We believe that our approach has significant potential applications to other types of man-machine dialogues, especially speech-understanding systems.

BIBLIOGRAPHY

Abeille, A., and Schabes, Y., (1989) Parsing Idioms in Lexicalized TAGs. *Proceedings of the Fourth Conference of the European Chapter of the Association for Computational Linguistics,* 10 - 12 April 1989, Manchester, England, 1-9.

Ahlswede, T., and Evens, M., (1988) A Lexicon for a Medical Expert System. In M. Evens, ed., *Relational Models of the Lexicon,* Cambridge University Press, Cambridge, England, 97-111.

Aho, A., and Ullman, J., (1972) *The Theory of Parsing, Translation and Compiling*, Volume I: *Parsing*. Prentice-Hall, Englewood Cliffs, New Jersy.

Aho, A., and Johnson, S., (1974) LR Parsing. *Computing Surveys,* Vol. 6, No. 2, 99-124.

Aho, A., and Ullman, J., (1973) *The Theory of Parsing, Translation and Compiling*, Volume II: *Compiling*. Prentice-Hall, Englewood Cliffs, New Jersy.

Aho, A., and Ullman, J., (1977) *Principles of Compiler Design*. Addison Wesley, Reading, Massachusetts.

Aho, A., Sethi, R., and Ullman, J., (1986) *Compilers, Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts.

Al-Khrisat, M., (1992) *Structuring the Arabic Lexicon and Thesaurus with Lexical-Semantic Relations to Support Information Retrieval*. Ph.D. Dissertation, Computer Science Department, Illinois Institute of Technology, Chicago, Illinois.

Allen, J., (1987) *Natural Language Understanding*. Benjamin/Cummings, Menlo Park, California.

Backhouse, R., (1979) *Syntax of Programming Languages: Theory and Practice*. Prentice-Hall International, London.

Baeza-Yates, R., and Gonnet, G., (1992) A New Approach to Text Searching. *Commun. ACM*, Vol. 35, No. 10 (Oct.), 74-82.

Bagley, R., (1992) *Mixed Mode Parsing with Noncanonical Extensions*. Ph.D. Dissertation, Computer Science Department, Illinois Institute of Technology, Chicago, Illinois.

Baker, L., (1993) *VHDL Programming with Advanced Topics*. John Wiley and Sons, New York.

Bentley, J., (1985) Programming Pearls: A Spelling Checker. *Commun. ACM,* Vol. 28, No. 5, (May) 94-102.

Berkel, B., and De Smedt, K. (1988) Triphone Analysis: A Combined Method for the Correction of Orthographical and Typographical Errors, *Second Conference on Applied Natural Language Processing,* February 1988, Austin, TX, 77-83.

Berwick, R., (1987) Intelligent Natural Language Processing: Current Trends and Future Prospects. In W. Grimson, R. Patil, eds., *AI in the 1980s and Beyond, An MIT Survey.* MIT Press, Cambridge, Massachusetts, 155-184.

Bresnan, J., (1987) *Lexical Functional Grammar*. *LI229*. Linguistic Summer Institute, Stanford University, Palo Alto, California.

Bresnan, J., and Kaplan, R., (1982) Lexical-Functional Grammar: a Formal System for Grammatical Representation. In J. Bresnan, ed., *The Mental Representation of Grammatical Relations*. MIT Press, Cambridge, Massachusetts, 173-281.

Carol, J., (1994) Relating Complexity to Practical Performance in Parsing with Wide-Coverage Unification Grammars. *Proceedings of the Association for Computational Linguistics,* June, Las Cruces, New Mexico, 287-294.

Carter, D., (1992) Lattice-Based Word Identification in CLARE. *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics*, June 28-July 2, Newark, Delaware, ACL, 159-166.

Chang, R., (1992) *Surface Level Generation of Tutorial Dialogue Using a Specially Developed Lexical Functional Grammar and Lexicon*. Ph.D. Dissertation, Computer Science Department, Illinois Institute of Technology, Chicago, Illinois.

Cormen, T., Leiserson, C., and Rivest, R. (1994) *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts.

Crane, H., and Rtischev, D., (1993) Pen and Voice Unite: Adding Pen and Voice Input to Today's User Interfaces Opens the Door for More Natural Communication with Your Computer. *Byte* (Oct.) 98-102.

Damerau, F., (1964) A Technique for Computer Detection and Correction of Spelling. *Commun. ACM*, Vol. 7, No. 3, 171-176.

Damerau, F., (1990) Evaluating Computer-Generated Domain-Oriented Vocabularies. *Information Processing & Management*, Vol. 26, No. 6, 791-801.

Damerau, F., and Mays, E., (1989) An Examination of Undetected Typing Errors. *Information Processing & Management*, Vol. 25, No. 6, 659-664.

De Heer, T., (1982) The Application of the Concept of Homeosemy to Natural Language Information Retrieval. *Information Processing & Management*, Vol. 18, No. 5, 229-236.

Diehl, S., (1994) Desktop Dictation. *Byte* (May) 145-146.

Durham, I., and Lamb, D., and Saxe, J., (1983) Spelling Correction in User Interfaces. *Commun. ACM*, Vol. 26, No. 10, (October), 764-773.

Earley, J., (1970) An Efficient Context-Free Parsing Algorithm. *Commun. ACM*, Vol. 13, No. 2, (February) 94-102.

Elmi, M., and Evens, M. (1993) An Efficient Natural Language Parsing Method. *Proceedings of the Fifth Midwest Artificial Intelligence and Cognitive Science Conference*, April 1993, Chesterton, Indiana 6-10.

Evens, M., ed., (1988) *Relational Models of the Lexicon*. Cambridge University Press, Cambridge, England.

Fischer, C., and LeBlanc. R., (1988) *Crafting a Compiler*. Benjamin Cummings Publishing, Menlo Park, CA.

Fox, B., (1991) Cognitive and Interactional Aspects of Correction in Tutoring. In P. Goodyear, ed., *Teaching Knowledge and Intelligent Tutoring.* Ablex Publishing Corporation, Norwood, NJ, 149-172.

Frederking, R., (1988) *Integrated Natural Language Dialogue*. Kluwer Academic Publishers, Boston, Massachusetts.

Glass, M., (1994) Circsim-Tutor Answer Grammar. Unpublished paper. Computer Science Department, Illinois Institute of Technology, Chicago, Illinois.

Granger, R., Staros, C., Taylor, G., and Yoshii R., (1983) Scruffy Text Understanding: Design and Implementation of the NOMAD System. *Conference on Applied Natural Language Processing*, February 1983, Santa Monica, CA, 104-106.

Hall, P., and Dowling, G., (1980) Approximate String Matching. *Computing Surveys,* Vol. 12, No. 4, (December) 381-402.

Herman, W., (1986) *The Portable English Handbook*, Third Edition. Harcourt Brace Jovanovich College Publishers, Orlando, Florida.

Holub, A., (1990) *Compiler Design in C*. Prentice-Hall International, London.

Jensen, K., and Heidorn, G., (1983) The Fitted Parse: 100% Parsing Capability in a Syntactic Grammar of English. *Conference on Applied Natural Language Processing*, February 1983, Santa Monica, California, 93-98.

Johnson, S., (1978) YACC: Yet Another Compiler Compiler. *Unix Programmers Manual,* Seventh edition, Volume 2, CBS College Publishing, New York, 353-387.

Kaplan, R., (1989) The Formal Architecture of Lexical-Functional Grammar. *Journal of Information Science and Engineering*, No. 5, 305-322.

Kaplan, R., Netter, K., Wedekind, J., and Zaenen, A., (1989) Translation by Structural Correspondences, *Fourth Conference of the European Chapter of the Association for Computational Linguistics*, Association for Computational Linguistics, Morristown, NJ. 272-281.

Keene, S., (1989) *Object-Oriented Programming in Common Lisp, A Programmer's Guide to CLOS*. Addison Wesley, Reading, Massachusetts.

Kim, N., Evens, M., Michael, J., and Rovick, A., (1989). CIRCSIM-TUTOR: An Intelligent Tutoring System for Circulatory Physiology. In H. Mauer, ed., *Computer Assisted Learning, Proceedings of the International Conference on Computer Assisted Learning*, Dallas, Texas. Springer-Verlag, 254-266.

Knuth, D., (1973) *The Art of Programming.* Vol. 3, *Sorting and Searching.* Addison Wesley, Reading, Massachusetts.

Kukich, K., (1992a) Spelling Correction for the Telecommunications Network for the Deaf. *Commun. ACM*, Vol. 35, No. 5, (May), 80-90.

Kukich, K., (1992b) Techniques for Automatically Correcting Words in Text. *ACM Computing Surveys*, Vol. 24, No. 4, (December), 377-439.

Lee, J., (1983a) Programming Linguistics. In A. Ralston, E. Reilly ed., *Encyclopedia of Computer Science and Engineering*, Van Nostrand Reinhold Company, New York, 1232-1243.

Lee, J., (1983b) Syntax, Semantics, and Pragmatics. In A. Ralston, E. Reilly ed., *Encyclopedia of Computer Science and Engineering*, Van Nostrand Reinhold Company, New York 1473-1474.

Lee, Y., (1990) *Handling Ill-Formed Natural Language Input for an Intelligent Tutoring System*. Ph.D. Dissertation, Computer Science Department, Illinois Institute of Technology, Chicago, Illinois.

Lesk, M., and Schmidt, E., (1978) Lex -A Lexical Analyzer Generator, *Unix Programmers Manual,* Seventh edition. Volume 2, CBS College Publishing, New York, 388-400.

Lewis, P., Rosenkrantz, and D., Stearns, R., (1976) *The System Programming Series, Compiler Design Theory*. Addison Wesley, Reading, Massachusetts.

MacLachlan, R., (1992) *CMU Common Lisp User's Manual.* School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Malloy, R., (1994) First Looks, Voice Recognition: IBM's Dictation System Is Quite a Talk Show. *PC Magazine*, April 1994 Volume 13 Number 7, 52.

Marsh, E., (1983) Utilizing Domain-Specific Information for Processing Compact Text. C*onference on Applied Natural Language Processing*, February 1983, Santa Monica, California, 99-103.

Mason, T., and Brown, D., (1990) *Lex & Yacc.* O'Reilly & Associates, Inc. Sebastopol, California.

Mayer, G., (1992) *Creating a Structured Knowledge Base by Parsing Natural Language Text*. Ph.D. Dissertation, Computer Science Department, Illinois Institute of Technology, Chicago, Illinois.

McIlroy, M., (1982) Development of a Spelling List. *IEEE Trans. Commun.* Vol. COM-30, No. 1, (Jan.), 91-99.

Meisel, W., (1993) Talk to Your Computer: Voice Technology Lets You Verbally Command Your Computer or Convert Speech to Text. *Byte* (Oct.), 113-120.

Miller, G., (1995) WordNet: A Lexical Database for English. *Commun.* ACM vol.38, No. 11, (November), 39-41.

Mitton, R., (1987) Spelling Checkers, Spelling Correctors, and the Misspellings of Poor Spellers. *Information Processing & Management.* Vol. 23, No. 5, 495-505.

Peterson, J., (1986) A note on Undetected Typing Errors. *Commun.* ACM vol.29, No. 7, (July), 633-637.

Pin-Ngern, S., (1990) *A Lexical Database for English to Support Information Retrieval, Parsing, and Text Generation*. Ph.D. Dissertation, Computer Science Department, Illinois Institute of Technology, Chicago, Illinois.

Pollock, J., and Zamora, A., (1984) Automatic Spelling Correction in Scientific and Scholarly Text. *Commun.* ACM vol.27, No. 4, (Apr.), 358-368.

Quirk, R., Greenbaum, S., Leech G., Svartvik J., (1985) *A Comprehensive Grammar of the English Language*. Longman Publishing Group, New York.

Ralston, A., and Lee, J., (1983) Parsing. In A. Ralston, E. Reilly ed., *Encyclopedia of Computer Science and Engineering*, Van Nostrand Reinhold Company, New York, 1105-1106.

Ramachandran, K., (1994) *Lexical Choice in Natural Language Text Generation for an Intelligent Medical Tutoring System*. M.S. Thesis. Computer Science Department, Illinois Institute of Technology, Chicago, Illinois.

Read, W., Quilici, A., Reeves, J., Dyer, M., and Baker, E., (1988) *Natural Language Systems Sourcebook, Artificial Intelligence Measurement Systems (AIMS) Project*, Review draft. Computer Science Department, University of California, Los Angeles, California.

Revri, A., (1987) *Handling Ill-Formed Input for Natural Language Interfaces*. M.S. Thesis, Illinois Institute of Technology, Chicago, Illinois.

Riseman, E., and Hanson A., (1974) A Contextual Postprocessing System for Error Correction Using Binary n-Grams. *IEEE Trans. on Computers. Vol.* C-23, No. 5, (May), 480-493

Roe, D., (1993) Voice Recognition over the Telephone. *Byte* (October), 114.

Rudnicky, A., Hauptmann, A., and Lee, K., (1994) Survey of Current Speech Technology. *Commun. ACM*, Vol. 37, No. 3 (March), 52-57.

Sager, N., (1981) *Natural Language Information Processing, A Computer Grammar of English and Its Applications*. Addison Wesley, Reading, Massachusetts.

Sells, P., (1986) *Lectures on Contemporary Syntactic Theories: An Introduction to Government-Binding Theory, Generalized Phrase Structure Grammar, and Lexical Functional Grammar*. CSLI University of Chicago Press, Chicago.

Sethi, R., (1989) *Programming Languages: Concepts and Constructs*. Addison Wesley, Reading, Massachusetts.

Seu, J., (1992) *The Development of an Input Understander for an Intelligent Tutoring System Based on a Sublanguage Study*. Ph.D. Dissertation, Computer Science Department, Illinois Institute of Technology, Chicago, Illinois.

Shapiro, S., (1983) Natural Language Processing. In A. Ralston, E. Reilly, eds., *Encyclopedia of Computer Science and Engineering*, Van Nostrand Reinhold Company, New York, 1101-1016.

Steele, G., (1984) *Common Lisp, Reference Manual.* Digital Press, Bedford, Massachusetts.

Steele, G., (1990) *Common Lisp, The Language*, Second Edition. Digital Press, Bedford, Massachusetts.

Tomita, M., (1986) *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers. Boston, Massachusetts.

Tomita, M., (1987) An Efficient Augmented-Context-Free Parsing Algorithm. *Computational Linguistics*, Vol. 13, Nos. 1-2, January-June, 31-46.

Tomita, M., (1988) Graph-Structured Stack and Natural Language Parsing. *Proceedings of 26th Annual Meeting of the Association for Computational Linguistics*, 7-10 June, 249-255.

Veronis, J., (1988) Morphosyntactic Correction in Natural Language Interfaces. *Proceedings of the 12th International Conference on Computational Linguistics* (Budapest, Hungary), 708-713.

Webster's Ninth New Collegiate Dictionary, (1988) Merriam-Webster Inc., Publishers, Springfield, Massachusetts.

Webster's Encyclopedic Unabridged Dictionary of the English Language, (1989) Gramercy Books, New York.

Weischedel, R., (1983) Handling Ill-Formed Input: Session Introduction. *Conference on Applied Natural Language Processing*, February 1983, Santa Monica, CA, 89-92.

Wescoat, M., (1987) *Practical Instructions for Working with the Formalism of Lexical Functional Grammar*. In Bresnan, J., ed., *Lexical Functional Grammar. LI229*. Linguistic Summer Institute, Stanford University, 4-37.

Winograd, T., (1983) *Language as a Cognitive Process,* Volume I*: syntax*. Addison Wesley, Reading, Massachusetts.

Wu, S., and Manber, U., (1992) Fast Text Searching Allowing Errors. *Commun. ACM*, Vol. 35, No.10 (October), 83-91.

Yannakoudakis, E., and Fawthrop, D., (1983) The Rules of Spelling Errors. *Information Processing & Management*, Vol.19, No. 2, 87-99.

Younger, D., (1967) Recognition and Parsing of Context-Free Languages in Time n^3. *Information and Control,* Vol. 10, No. 2, 189-208.

Zue, V., (1987) Automatic Speech Recognition and Understanding. In W. Grimson, R. Patil, eds., *AI in the 1980s and Beyond, An MIT Survey,* MIT Press, Cambridge, Massachusetts, 185-200.