CS–1993–15

Design and Development of Spoken Natural-Language Dialog Parsing Systems

Dwayne Richard Hipp

Department of Computer Science

Duke University

Durham, North Carolina 27708-0129

June 21, 1993

Design and Development of Spoken Natural-Language Dialog Parsing Systems

Dwayne Richard Hipp

June 21, 1993

Supervised by Alan W. Biermann

Dissertation submitted in partial fulfillment
of the requirements for the degree
of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

This document is a reformatted version of the disseration, and equivalent in content.

Abstract

The development of a practical spoken language dialog system is fraught with difficulties. This thesis describes the following three problems relating to the natural language parsing and presents a workable solution for each:

1. The parser in a voice dialog system must contend with ill-formed inputs. Because the parser receives its input from a speech recognizer, the input will often contain misrecognitions. Also, the user's original speech may be elliptical, telegraphic, or otherwise grammatically ill-formed.

   The thesis describes a new parsing strategy aimed at overcoming the ill-formedness of dialog speech. A new algorithm for *minimum-distance parsing* is used to decode the structure in the user's utterance, and *syntax-directed translation* is used to convert the utterance into an internal representation. Finally, *dialog expectation* is used to resolve ambiguity, anaphora, and ellipsis in the input utterance. In experimental tests, the new parser found the correct interpretation for 83% of 2804 utterances even though only 48% of those utterances were syntactically well-formed.

2. The development of a natural language grammar for a speech dialog system becomes increasingly difficult as the size of the grammar increases.

   This thesis describes several new algorithms collected into an integrated grammar development system which simplifies the task of developing a large natural language grammar. This system allows the user to focus on the small subset of the grammar which is of immediate interest, and thereby allow for faster, easier, and more error free editing. Diagnostic and performance information may also be obtained.

3. Regardless of the parsing strategy used, the parser will sometimes make mistakes.

   This thesis describes a subsystem which uses verification subdialogs to selectively verify meanings output by the parser. The propensity of the system to verify a given input is easily adjusted so as to provide any required level of effective parsing accuracy. Experimental data shows that verification can raise the effective accuracy of the parser from 83% to 97% without excessive burden to the user.

Acknowledgements

This dissertation was made possible by the gracious cooperation and assistance many individuals. Most notably, Dr. Alan W. Biermann inspired and directed the research which lead to this thesis. I am deeply in his debt for his patient and insightful review of the many drafts of this dissertation, and of the entire experimental program from which this dissertation grew.

Dr. Ronnie W. Smith also played a major and important role in this work. He wrote most of the code for the dialog controller and other higher-level intellengence in the Circuit Fix-it Shoppe program, then painstakingly integrated his system with my parser to make the whole system run. With help from Dania M. Egedi, Ronnie conducted the gruelling series of user experiments from which most of the performance data herein reported has been derived.

Countless others have contributed to the dissertation in smaller, but no less important, ways. Dr. Robert D. Rodman at North Carolina State University has been an important collaborator throughout the project. The Duke computer science department's Lab Staff played a critical role by providing us with fast, reliable hardware and outstanding systems support. The comments, suggestions, and encouragement of others, too numerous to mention, have also been most helpful in keeping this project going.

Finally, the research represented in this dissertation would not have been possible but for the financial support of the National Science Foundation through grant number NSF-IRI-88-03802 and a grant from Duke University which was graciously arranged by Dr. Charles Putman.

Contents

List of Figures

List of Tables

Chapter 1

Background

The one feature of man that enables him to excel above other species is his skill in the use of language. So deft is he with words, that he tends not to comprehend fully the the difficulty of abstract communication. This fact is nowhere more plainly seen than in the history of efforts to program computers to understand human language. It seems like it should be so simple. Do not even mentally crippled children learn to speak fluently in only a few years? And yet the problem of teaching human language to computers has resisted attack by the world's brightest minds for over four decades. Therefore, let the readers' expectations for this thesis be tempered.

But there is still reason for optimism. The speed and memory capacity of computing hardware has been increasing exponentially, and promises to continue to do so into the foreseeable future. With greatly increased computing resources and many new and innovative programming techniques the long elusive dream of a intelligent talking machine may someday become a reality.

This thesis represents one small step toward the ultimate goal of a machine which converses freely and naturally in a human language. Herein is described a new approach to the development of a system which can parse and understand ill-formed spoken natural language in the context of a dialog. A specific implementation of this new approach, when coupled to a task-oriented dialog controller (not a part of this thesis), performs remarkably well, and is in fact upon the verge of being a practical industrial device.

1.1 A Statement of the Problem which this Thesis Addresses

A block diagram of the input side of a typical *spoken language system* (SLS) is shown in figure 1.1. A *speech recognizer* listens to the sound of the user's speech and outputs one or more plausible transcriptions of what was spoken. The output of the speech recognizer feeds the *parser*. The parser will analyze the transcript of the user's speech and determine the meaning which the user wishes to communicate. The parser is aided by a *natural language grammar* which describes the syntax of the user's language, and the mapping from this language into the system's internal representation of meaning. Note that the natural language grammar is distinct from the grammar used to limit search in the speech recognizer. The output of the parser is sent to a *dialog controller*. The dialog controller implements some of the higher level cognitive functions of the SLS. One particular function of the dialog controller is to provide *dialog expectation* to the parser. Presumably, the dialog controller communicates with other components of the complete SLS, such as a natural language generator, a knowledge database, and some form of reasoning system; but as these other components do not play a role in this thesis, they are omitted from figure 1.1.

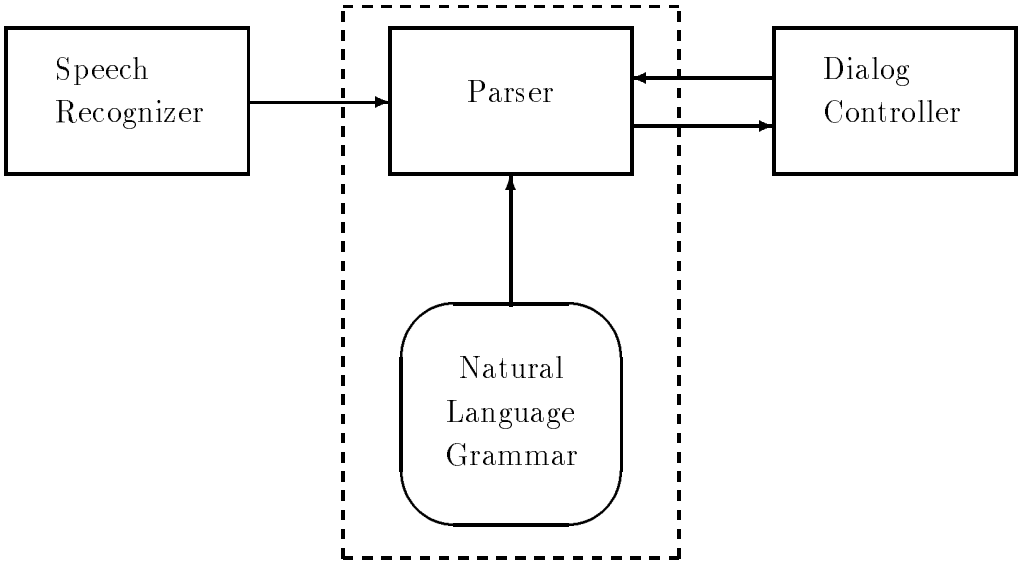

Figure 1.1: Block diagram of a typical spoken language system

1.2 Contributions of this Thesis

This thesis provides new information which is useful in the design of those parts of figure 1.1 which are enclosed in the dashed box: the parser and the natural language grammar. Four results are obtained.

1. A new parsing algorithm is described which is effective at parsing spoken natural-language inputs in the context of a dialog. Prior work on natural language parsers has primarily focused on parsing of text soliloquies. But text soliloquy has a very different structure from spoken dialog, and as a result, traditional parsing methods do not work adequately for spoken dialog. The differences between spoken dialog and text soliloquy, a summary of related prior research, and an overview of the new parsing algorithm are described in section 2.1. Details of the new parsing algorithm and experimental results from an implementation of this algorithm are given in chapter 3.

2. Algorithms and tools which are useful in the development of large natural language grammars are described. It is observed that the development of large grammars for systems which use natural language can potentially consume most of the human effort applied toward developing such systems. Automated grammar development aids are therefore desirable to expedite the development process. Section 2.2 further motivates the need for grammar development aids. That section also summarizes previous work on automated grammar development aids. Chapter 4 describes the more interesting algorithms in detail.

3. Methods by which the output of a natural language parser can be verified in order to improve the accuracy of the parser are also considered. It turns out that adding selective verification of meanings to a dialog processing system can greatly enhance the accuracy and performance of that system. Further motivation of the need for verification, and an overview of the results obtained, is given in section 2.3. Details of verfication techniques and measurements of performance improvements which these techniques give to a dialog processing system are

both described in detail in chapter 5. The idea of selective verification of meanings in a dialog processing system appears to be original, as no prior discussion of this idea could be found in the literature.

4. Finally, the thesis describes efficient, robust, portable, and public-domain implementations of the algorithms mentioned above. These implementations are effective tools which may be used unaltered in future research in natural-language dialog systems. They may serve as models for new and more powerful natural-language dialog system development tools which are designed in the future.

1.3 The Origin of this Thesis

This thesis has grown out of efforts to develop a voice dialog system called the "Circuit Fix-it Shoppe". The mission of the Circuit Fix-it Shoppe is to use voice interaction to assist people in the repair of an electronic circuit. Further details on the Circuit Fix-it Shoppe can be found in [36]. A 12-minute videotape describing and demonstrating this system has been prepared [18], and is included as an appendix to this thesis. The speech recognizer in the Circuit Fix-it Shoppe is a commercial speaker dependent connected speech recognizer with a 125 word vocabulary. The dialog controller was developed by Ronnie Smith and is the topic of his Ph.D. thesis [35]. Other parts of the Circuit Fix-it Shoppe program have been developed by Robin Gambill and Dania Egedi.

Chapter 2

Overview

This chapter gives an overview of each of the four main results obtained in this thesis and originally listed in section 1.2. This chapter might therefore be viewed as an amplification or enlargement of section 1.2. The intent of this chapter is to provide an overview of the problems addressed by this thesis, and their solutions, without burdening the reader with excessive detail. A detailed analysis of the problems and solutions appears in the chapters that follow.

2.1 The Parsing Problem

As is pointed out in [6], [9], [16], [40], and [43], the parser in an SLS is confronted with difficulties that are not encountered by traditional text-based natural language parsers.

1. The input to the system is *spoken*, and then interpreted by a speech recognizer. The speech recognizer will typically generate multiple interpretations of the user's speech none of which may be absolutely correct.

2. Input to the system may be *syntactically ill-formed*. Even in the absence of speech recognition errors, the input to the parser may not be a well-formed phrase according to the natural language grammar used by the parser. Ill-formedness may have several causes:

   - Speech performance errors. The users will not always say exactly what they intended due to slips of the tongue, poor choice of words, or other problems.
   - Telegraphic speech. In normal conversation, structure words such as "is" or "to" are usually deemphasized. When conversing with a computer with imperfect language skills, many people will subconsciously omit such structure words altogether, resulting in a phrase that sounds like a telegraph message. (See [14].)
   - Vocabulary errors. Users will sometimes speak words which are not in the vocabulary of either the speech recognizer or the parser.
   - Inadequacies in the language model. The system's understanding of the natural language used for input is probably incomplete and inaccurate.

3. Because the system operates in the context of a dialog, inputs may be *elliptical*. Many utterances will be sentence fragments. Others will contain pronouns with indefinite or ambiguous antecedents.

4. Finally, the meaning of inputs to the system tend to be highly *context dependent*. The same phrase spoken at different times can have divergent meanings. The particular meaning given to an input is strongly influenced by the preceding dialog.

Most modern speech recognition systems incorporate linguistic knowledge in order to reduce the amount of search. However, this linguistic information is often in the form of a finite-state graph or word-pair transition probability matrix and is not suitable for detailed linguistic analysis. This low-level linguistic knowledge is sometimes automatically generated from higher-level linguistic knowledge available in a downstream parser, as in [45]. Even among systems which use more sophisticated linguistic knowledge ([31], [40], [42]), there is still the implicit assumption that all inputs are syntactically well-formed. In some systems ([23] and [46] for example) the well-formedness assumption is made explicit. There are no speech recognition or speech understanding systems in current publication which even attempt to handle problems of ellipsis and context dependency.

Various efforts have been made to construct natural language parsers which can understand syntactically ill-formed inputs and which are not integrated with a speech recognizer. A method for parsing a context-free language with an unknown number of missing words is described in [24], but this method's usefulness is limited by the necessity of knowing where the missing words should be. Case frame and island parsing techniques are described in [13], [16], [17], [39], and [40]. These techniques are effective when the ungrammatical sections of an input occur in between grammatically well-formed phrases, but have difficulty when important phrases are themselves ill-formed. A system described in [11] uses dialog expectation to repair ill-formed inputs. Some advocate a connectionist approach, as in [20] and [38], but the effectiveness of these techniques has not yet been demonstrated.

Prior work on methods for handling ellipsis and context dependent meaning has mostly studied text input, which is not subject to the distortions and errors associated with speech input. Summaries of this prior work can be found in [3], [12] and [34].

2.1.1 A New Solution

Chapter 3 of the thesis will describe a new parsing strategy for ill-formed spoken dialog. This new parser has three major subcomponents.

- *Minimum-distance parsing* (MDP) is used to decode the underlying structure of each input utterance.
- *Syntax-directed translation* (SDT) is used to translate the utterance into a Prolog predicate which encapsulates the meaning which the user intended to communicate.
- Finally, *dialog expectation* is used to resolve ellipsis and anaphora and to provide dialog context to disambiguate an otherwise ambiguous utterance.

2.1.2 Prior Art

Basic minimum-distance parsing algorithms are described in [2], [26], and [27]. In [2], Aho and Peterson describe a two-step method for minimum-distance parsing. First the grammar is augmented with extra rules called *error productions*. Then a parser is described which uses the minimum number of error productions. The Aho and Peterson algorithm runs in time $O(n^3k^2)$ where n is the number of symbols in the string to be parsed and k is the number of symbols in the original unaugmented grammar. In [27], Lyon presents a one-step algorithm for minimum-distance parsing which is based on Earley's generalized parsing algorithm [8], and which also runs in time $O(n^3k^2)$.

Levinson in [26] describes a minimum-distance parsing algorithm based on Younger's generalized parsing method [44]. However, Levinson's algorithm requires that the grammar be in Chomsky normal form, which precludes its use for syntax-directed translation, since syntax-directed translation grammars cannot in general be put in Chomsky normal form[1]. Levinson suggests the use of his algorithm for speech recognition, but states that no one has ever before used minimum-distance parsing of a context-free language for that purpose (though many have used minimum-distance parsing of regular languages.) The first reported use of Levinson's algorithm in a speech recognition application is by [29].

Theoretical issues of syntax-directed translation were summarized by Aho and Ullman in [1].

The author is aware of no prior work which uses dialog expectation in any manner resembling its use in the new parser.

2.1.3 An Intuitive Description of the Solution

The *distance* between two strings of words is defined to be the minimum weighted sum of the number of insertions and deletions needed to convert one string into the other. The distance is intended to be an approximate measure of the difference in meaning between the two strings. That is, a large distance should indicate that the two strings mean very different things, but a small distance should show that the strings have a similar (or identical) meaning. For this reason, words such as "not" which make a big difference in the meaning of an utterance if they are inserted or deleted are given a large weight, whereas words like "the" which do not significantly alter the meaning of an utterance if inserted or deleted are given a small weight.

The parser has two inputs: a *syntax-directed translation grammar* (SDTG) and the string of words spoken by the user. The SDTG defines a context-free language called the *input language* and a meaning for every string in the input language. The goal of the algorithm is to find an appropriate meaning for the user's words. When the user's input happens to correspond exactly to some string in the input language the task is simple – the algorithm just finds the meaning associated with the input language string and that is also the meaning of the user's input. But if the user's input is not in the input language, the situation is more troublesome. This situation is where the "minimum-distance" aspect of minimum-distance parsing comes into play. Since the spoken utterance itself is not in the input language, the algorithm finds a string which *is* in the input language and which is closest (or has minimal distance to) the user's string of spoken words, and uses the meaning of this closest string as the meaning of the user's input. The hope is that the user's input and the string in the input language are sufficiently close that they share the same meaning.

In the above discussion, and throughout this thesis, a *meaning* is simply a string in a context-free language, called the *output language*, which is defined by the SDTG. In the implementation, the output language happens to consist of a class of Prolog structures which are understood by the dialog controller. Other implementations, however, might well define very different output languages. The mapping between strings of the input language and strings of the output language is computed by the SDT algorithm in accordance with the SDTG. The SDT component of the parser is combined with MDP to produce a new algorithm called *minimum-distance syntax-directed translation* (MDSDT).

There are two complications to the parsing system described above. First, the input to the parser is not a single string of words, but a *lattice*. A lattice is an acyclic directed graph with a

[1] Proof: An exhaustive search will show that nonterminals on the right-hand side of the syntax-directed translation production rule "$S \rightarrow ABCD : BDAC$" cannot be grouped together to yield only two right-hand side nonterminals without disrupting the mapping to the nonterminals in the semantic part of the production rule.

single source and a single sink. Each arc of the lattice is labeled with a word, so that the collection of all paths through the lattice beginning at the source and ending at the sink is a set of many strings of words, any of which might be what the user actually said. Thus a lattice is a compact representation of a large set of possible user inputs. The MDP algorithm which accepts a lattice as input is an intuitively straightforward extension of MDP which inputs a solitary string. The lattice-input MDP finds a string in the CFG's language which is closest to any of the strings represented by the lattice and uses that string to find the meaning of the user's input.

The second complication is that two or more strings in the input language may be at the same distant from the user's utterance, or else the difference in distances may be slight. In such cases, it is necessary to find multiple meanings for the user's utterance based on all proximate strings in the input language and then to select one of these possible meanings. The selection of one of several possible meanings for an utterance is the principal use of dialog expectation.

Dialog expectation is a set of meanings (that is to say, a set of strings in the SDTG's output language) which are generated by the dialog controller. Each meaning in this set is called an *expectation*. Each expectation comes with an *expectation cost* which is a measure of how strongly the expectation is anticipated at the current point in the dialog. When the MDP algorithm outputs multiple meanings, each of the meanings has an associated *utterance cost* which is just the distance between the user's input and the string from the input language which generated the meaning. Each meaning is matched with its corresponding dialog expectation, and the corresponding expectation and utterance costs are combined into a *total cost* by an *expectation function*. The meaning with the smallest total cost is selected to be the final output of the parser. An important side-effect of matching MDP meanings with expectations is that ellipsis and anaphora in the user's speech are often resolved based on what the dialog controller expected the user to say.

2.1.4 Experimental Results

The new parser has been tested in an actual SLS. Using subjects that have no special knowledge of computers or the implemented SLS, the parser was able to find the correct meaning for 83% of 2804 inputs even though only 48% of these inputs were grammatical.

Several different expectation functions were tested and it was found that the one which gave the best results used the expectation cost only to break ties between meanings with the very best utterance cost. This is a much weaker use of expectation cost than was anticipated.

2.2 Grammar Development

Beesley and Hefner [4] claim that 80% of the development time for an SLS is typically spent working on the natural language grammar. Empirical observations of the time spent developing the 500 rule grammar for the Circuit Fix-it Shoppe are consistent with Beesley and Hefner's claim.

In spite of the immoderate amount of time required to develop a natural language grammar, comparatively little research has been directed toward developing tools or techniques which can assist in grammar development. Some *software aided grammar development systems* (SAGDSs) have been described. Many of these SAGDSs provide an interactive environment which allows breakpoints to be set in the grammar for testing and tracing purposes [4] [15] [22] [41]. One SAGDS, described in [5], provides the ability to restrict attention to a small subset grammar. Memmi and Mariani [28] describe an SAGDS which will generate random sentences from the grammar under development, and generate a complete lexicon for the grammar.

The following categorization of the grammar development problem is based on the experience of developing a 500-rule grammar for the Circuit Fix-it Shoppe.

1. *Correcting errors.* Some well-formed phrases in the input language will be translated into the wrong output language string. When these errors are found they must be corrected.

2. *Making incremental changes and additions.* During the development of the SLS, the language output by the parser and input by the dialog controller will occasionally be modified to accommodate unforeseen semantic representation problems. Additions will also have to be made to the input language of the parser as the developers discover that users sometimes phrase their statements in ways which were not initially anticipated.

3. *Testing for errors and adequate language coverage.* The grammar should be extensively analyzed for errors of commission as will as omission. The following are some, but not all, of the kinds of tests that are desirable:

   - Is every rule in the grammar actually used? Unused rules might be a vestige of some obsolete functionality, but they might also indicate that some part of the grammar was accidently left in an incomplete state.
   - Does the input terminal set correspond to the vocabulary of the speech recognizer?
   - Are all outputs of the parser well-formed from the point of view of the dialog controller?
   - How many different ways can one communicate a given meaning to the parser?

There are many techniques which can be useful for correcting errors, making incremental changes, and testing in a large natural language grammar. In the sequel, emphasis is placed on methods suitable for use with SDTGs, though many of the techniques described may be useful with ordinary context-free grammars as well.

1. Techniques for extracting small, manageable subset grammars from a large unwieldy parent grammar.

   Operations of this type are useful when studying or modifying some specific aspect of the grammar. When an analysis of a large grammar is attempted manually, the analyst tends to devote a preponderance of time searching through the grammar for those few production rules which are relevant to the feature or behavior being studied. Not only is this process slow and tedious, but it is also error prone. One is never quite sure if all pertinent production rules have been found.

   With the aid of tools which will automatically search for specific production rules, the task of testing or enhancing some specific behavior of the grammar is made much easier. Operations which are useful automatic search aids include:

   - Finding the union, intersection, or difference between two subsets of a grammar. These kinds of operations are used to combine the results of operations described in subsequent bullets.
   - Finding the subset of rules in a grammar which can be derived from, or are derived by some other rule subset of that grammar. Operations of this kind are useful for adding details to a grammar subset found by operations of the next two bullets.
   - Finding the subset of rules in a grammar which match a given regular expression. This operation is useful for finding one or more production rules relating to a single word or concept.

- Finding the subset of rules in a grammar which can be used to recognize (generate) any string in the intersection of the input (output) language of a SDTG and the language of a given regular expression. The great usefulness of this operation is described in section 4.2.

2. Performance evaluation techniques.

   Whether one is trying to isolate an error, or verify that a recent change has the intended effect, or scan for inadequencies in the language model, grammar analysis tools will make the job much faster and less error prone. Performance analysis tools might include:

   - The ability to parse a chosen input. One should be able parse using any subset of the grammar. Also, it is often useful to be able to root the parse on some nonterminal other than the usual starting nonterminal, and thereby parse a subphrase which is not itself a complete well-formed input.
   - The ability to produce a parse tree for any phrase. As with a direct parse, one should be able to do this using a subset grammar, and starting at any nonterminal.
   - The ability to list and count all well-formed strings in some subset of either the input or output language. This operation is useful for testing the coverage of the input grammar. It is also useful in determining whether or not a grammar subset will generate any unexpected meanings. Since both input and output languages will probably be infinite, it is necessary to restrict the listing in some way.
   - The ability to list or count all terminals, nonterminals, or rules in the grammar or any subset of the grammar. These operations are useful to verify that the grammar is both self-consistent and consistent with what the speech recognizer generates and what the dialog controller expects to see. These operations can also serve as general utilities in an integrated grammar development package.

3. Techniques for extending, modifying, and revising the grammar.

   For this thesis operations in this category are limited to editing the text of a grammar subset using a standard text editor. Future work may include automatic inference of new production rules based on a set of examples and counter-examples, or automatic simplification and consolidation of production rules within an existing grammar.

Chapter 4 details algorithms for implementing the development tools described above. The grammar development techniques described above have been integrated into a complete SAGDS which is described in Chapter 6 It is believed that this new SAGDS will be a significant help in developing new natural language grammars, and that it would have greatly reduced the time needed for the development of the Circuit Fix-it Shoppe's grammar had it been available when that grammar was developed.

2.3 Verification subdialogs

Chapter 5 of the thesis describes techniques for verifying the meaning of what a user says when there is doubt that the meaning has been correctly deduced.

Every natural language parser will sometimes misunderstand its input. Misunderstandings can arise from speech recognition errors or inadequacies in the language grammar, or they may result from an input which is ungrammatical or ambiguous. Whatever their cause, misunderstandings

can jeopardize the success of the larger system of which the parser is a component. For this reason, it is important to reduce the number of misunderstandings to a minimum.

In a dialog system, it is possible to reduce the number of misunderstandings by requiring the user to verify each utterance. Some speech dialog systems implement verification by requiring the user to speak every utterance twice, or to confirm a word-by-word readback of every utterance. Such verification is effective at reducing errors which result from word misrecognitions, but does nothing to abate misunderstandings which result from other causes. Furthermore, verification of all utterances can be needlessly wearisome to the user, especially if the system is working well.

A superior approach is to have the SLS verify the deduced meaning of an input only under circumstances where the accuracy of the deduced meaning is seriously in doubt, or correct understanding is essential to the success of the dialog. The verification is accomplished through the use of a *verification subdialog*, which is a short sequence of conversational exchanges intended to confirm or reject the hypothesized meaning. The following example of a verification subdialog will suffice to illustrate the idea:

computer:	What is the LED displaying?
user:	The same thing.
computer:	Did you mean to say that the LED is displaying the same thing?
user:	Yes.

As will be seen in chapter 5, selective verification via a subdialog results in an unintrusive, human-like exchange between user and machine.

A subsystem is described which has been added to the Circuit Fix-it Shoppe dialog system and which uses a verification subdialog to verify the meaning of the user's utterance only when the meaning is in doubt or when accuracy is critical for the success of the dialog. Notable features of this verification subsystem include:

- Verification is *selective*. A verification subdialog is initiated only if there is reason to suspect that the overall performance and accuracy of the dialog system will be improved. In this way, the verification subsystem responds much as a person would.

- Verification is *tunable*. The propensity of the system to verify can be adjusted so as to provide any required level of speech understanding accuracy.

- Verification *operates at the semantic level*. The system verifies an utterance's meaning, not its syntax. This helps overcome misunderstandings which result from inadequacies in the language model, or ungrammatical or ambiguous inputs.

The most important aspect in the development of a verification subdialog system is the decision of whether or not a particular input should be verified. The system described in this thesis computes for each meaning a *confidence*, which is a measure of how plausible the parser's output is, and a *verification threshold* which is a measure of how important the meaning is toward the success of the dialog. A verification subdialog is undertaken if the confidence drops below the verification threshold. Details of how confidence and verification thresholds are computed form a major part of chapter 5.

Other factors in the development of a verification subdialog system include the algorithm used to implement subdialogs and the manner in which verification is added to the larger SLS. These issues are also treated in chapter 5.

The performance of the implemented verification system is aesthetically pleasing. The verification subdialogs are fluid and natural, so much so that some observers fail to notice that a

verification has taken place. Furthermore, the net accuracy of the parser is greatly improved. The percentage of utterances which are correctly understood can be raised from 83% to about 97% with only a small verification overhead.

2.4 Implementations

Chapters 3, 4, and 5 are primarily concerned with theoretical aspects of parsing, SAGDSs, and verification. But these things are not just theoretical concepts; they have all been implemented into working programs. Chapter 6 of this thesis describes these implementations. The reason for describing the implementations is three-fold:

1. It is hoped that the programs written in support of this thesis will be useful for future research both by the author and by others. To this end, the programs have been designed and written in a very portable manner so that they may be connected to diverse and divergent SLSs with a minimum of rework. Chapter 6 is intended to provide a *Users Reference Manual* for the implementions.

2. Some of the implementation techniques used are novel and have worked very well in that they have provided a stable and robust structure to the code which is also easily altered and modified. This is especially the case with the parser which is implemented as 6 separate processes under UNIX which communicate through pipes. Future researchers may wish to imitate these techniques even if they don't use the actual code.

3. Sometimes it is easier to understand abstract theory if one has a clear vision of the end result. Some readers may wish, therefore, to review the more concrete material in chapter 6 before perusing the earlier chapters.

Chapter 3

Parsing

This chapter reports on the new parser. The minimum-distance syntax-directed translation (MDSDT) algorithm, which forms the heart of the new parser, is described in section 3.3. The sections before 3.3 discuss various subproblems and simplifications of MDSDT in an effort to introduce the algorithm slowly and to make the presentation more readable and understandable. The parser's use of dialog expectation is described in section 3.4. Finally, section 3.5 reports on how an implementation of the new parser performed in actual experiments.

3.1 A New Algorithm for Minimum-distance Parsing

This section will describe a new algorithm for minimum-distance parsing (MDP) based on a depth-first search of a structure called the *parsing graph*. This new algorithm improves upon algorithms previously described in [2], [26], and [27] in two ways.

1. This algorithm will accept a lattice as input. Prior minimum-distance parsing algorithms accept only strings of symbols.

2. This algorithm is able to prune the search so as to be a constant factor faster than any previously reported minimum-distance parsing algorithm.

Both of these improvements are omitted from the initial description of the new MDP algorithm in order to make the initial description more lucid.

The reader should note that the MDP algorithm is not used directly. Rather, it is combined with syntax-directed translation, described in section 3.2, to produce the MDSDT algorithm described in section 3.3. The MDP algorithm is here presented in isolation only as an aid to understanding MDSDT.

3.1.1 Intuitive Overview of MDP

There are two inputs to the MDP algorithm, a context-free grammar (CFG) and an *input phrase* which is a string of symbols from the terminal alphabet of the CFG. (The terminal alphabet for the implementation is the subset of English words which are understood by the speech recognizer.) The language specified by the CFG is called the *input language*. Between every string in the input language and the input phrase there is a distance which is defined to be the minimum number of word insertions and deletions needed to convert the input phrase into the string of the input language. The output of the MDP algorithm is the least of these distances.

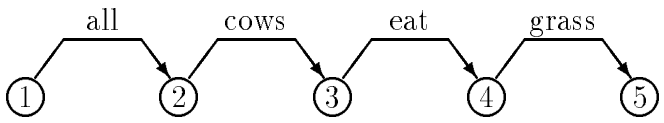

Figure 3.1: The input graph corresponding to the input phrase "all cows eat grass".

In practice, the MDP algorithm is modified slightly in order to compute the minimum *weighted* distance between the input phrase and the input language. The weighted distance is the sum of weights on each insertion and deletion needed to transform one string into the other. The weights are intended to show the relative importance of words to the meaning of the input phrase. Words such as "not" which make a big difference in the meaning of an utterance if they are inserted or deleted are given a large weight, whereas words like "the" which do not significantly alter the meaning of an utterance if inserted or deleted are given a small weight. In this way, the distance between two phrases is made a better estimate of the difference in their meanings. It is not difficult to weigh the insertions and deletions of distance computations differently, but it is yet another complication which makes the MDP algorithm more difficult to understand upon first reading. For this reason, the initial description of the MDP algorithm will consider insertions and deletions to be unweighted. The changes to the MDP algorithm which are necessary to implemented weighted insertions and deletions are summarized in section 3.1.10.

The input phrase is represented as a linear graph with each arc of the graph representing one word (or terminal symbol) of the input phrase. This graph is called the *input graph*. Suppose the input phrase is "all cows eat grass", then the corresponding input graph would be as shown in figure 3.1. The nodes of the input graph are numbered from 1 to V where node 1 is the *source* node and node V is the *sink* node. V is 5 in figure 3.1.

The MDP algorithm works by adding new arcs to the input graph. The added arcs may be either *nonterminal-arcs* (hereafter called NT-arcs) or they may be *rule-arcs* (called R-arcs). Each of the added arcs is marked with a distance. In order to distinguish them from the added arcs, the original arcs of the input graph are called *terminal-arcs* or T-arcs. As the names suggest NT-arcs are related to nonterminals in the CFG, T-arcs are related to terminals in the CFG, and R-arcs are related to rules of the CFG. The exact nature of these relationships will become evident presently.

In figure 3.1, all T-arcs span adjacent nodes. This is not necessarily the case for NT- and R-arcs. Later, the MDP algorithm will be generalized to allow T-arcs to span nonadjacent nodes as well.

The output of the MDP algorithm is the distance on the NT-arc spanning nodes 1 to V of the input graph and corresponding to the starting nonterminal of the CFG. Call this arc the *root arc* of the input graph. The distance on the root arc depends upon distances of other added arcs in the input graph. (A detailed description of how the distance is computed is given below.) The distances on these other arcs depend in turn on yet other arcs, and so forth, until ultimately the distance of the root arc depend upon the presence of T-arcs in the input graph.

It is conceptually helpful to construct a directed graph showing the dependency relationships of arcs in the input graph. Call this dependency graph the *parsing graph*. Each arc in the input graph corresponds to a node of the parsing graph. If A and B are two arcs of the input graph and the distance of A depends directly on the distance of B, then in the parsing graph there is an edge from node A to node B. B will be called a *child* of A. Call the node of the parsing graph which corresponds to the root arc of the input graph the *root node*. It will be shown below that the MDP algorithm is a depth-first search of the parsing graph beginning at the root node.

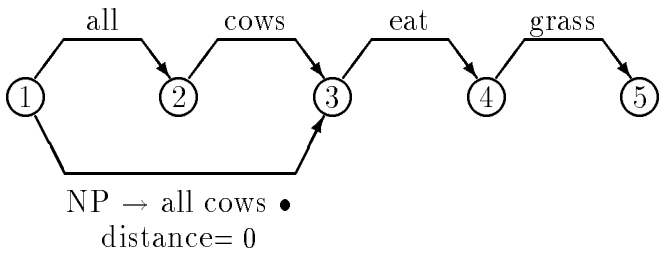

Figure 3.2: The R-arc $(r_x, 2, 1, 3)$ added to the input graph shown in figure 3.1.

3.1.2 Formal Notation

A context-free grammar or CFG is the 4-tuple $(\mathcal{N}, \Sigma, P, S)$ where $\mathcal{N}$ is a set of nonterminal symbols, Σ is a set of terminal symbols, $P = \{r_1, r_2, \ldots, r_{|P|}\}$ is a set of production rules, and $S \in \mathcal{N}$ is a special nonterminal which is the start symbol of the grammar. Each production rule, r_i, is a pair (n, α) where $n \in \mathcal{N}$ is the left-hand side (LHS), and $\alpha \in (\mathcal{N} \cup \Sigma)^*$ is the right-hand side (RHS). The notation $|r_i|$ means the same as $|\alpha|$, the number of symbols on the right-hand side of that rule. Assume, without loss of generality, that the nonterminal S does not appear on the RHS of any production rule. Further assume that every production rule which does not have S on its LHS has at least one symbol on its RHS. If a particular CFG does not meet these last two criteria, then it can easily be converted to a related CFG that does using methods outlined in, for example, [19].

A terminal-arc or T-arc is the triple (t, u, v) where $t \in \Sigma$ is a terminal symbol, and u and v are non-negative integer node numbers designating the head and tail nodes of an arc. A T-arc may be intuitively understood to represent a single word input to the parsing algorithm.

A rule-arc or R-arc is the quadruple (r, x, u, v) where r is a single rule in the CFG, x is an integer called the *index* such that $0 \leq x \leq |r|$ and, u and v are node numbers. Intuitively, each R-arc represents that the first x symbols on the RHS of rule r have been matched against some of the T-arcs between the nodes u and v. A distance associated with each R-arc gives the number of insertions and deletions needed to effect that match. Here are two simple examples of R-arcs: The R-arc $(r, 0, v, v)$ represents matching no symbols from the RHS of production rule r against no symbols of the input graphs. The R-arc $(r, |r|, 0, V)$ means that all symbols on the RHS of the production rule have been matched against all symbols of the input phrase. (V is the number of nodes in the input graph.) As a third example, suppose the production rule r_x is of the following form:

$$\text{NP} \rightarrow \text{all cows}$$

If the R-arc $(r_x, 2, 1, 3)$ is now added to the input graph of figure 3.1, the result is figure 3.2. The distance on this arc is 0 because the first two symbols on the RHS of the production rule exactly matching the T-arcs between nodes 1 and 3.

A nonterminal-arc or NT-arc is the triple (n, u, v), where n is a single nonterminal symbol, and u and v are node numbers. Intuitively, each NT-arc represents that the subset of rules in P which have the nonterminal symbol n as their LHS have been matched against the sequence of T-arcs between the nodes u and v. A distance associated with each NT-arc gives the minimum number of insertions and deletions needed to effect such a match.

3.1.3 Computation of Distance for R-arcs

The distance on an R-arc is computed recursively from the distances of other R-arcs. Specification 3.1 below is a precise description of how the distance on an R-arc is computed. This method for computing the distance on an R-arc is essentially the minimum edit-distance algorithm described in [37]. That the correct minimum distance is computed can be shown by induction.

Specification 3.1 *The computation of distance for the R-arc* (r, x, u, v).

1. If $x = 0$ and $u = v$ then the R-arc matches nothing against nothing, and its distance is therefore 0. This case is the basis for the recursion.

2. If $x > 0$ and $v > u$ and the T-arc between nodes $v - 1$ and v is labeled by the x-th symbol on the RHS of production rule r, then the distance on the R-arc (r, x, u, v) is the same as the distance on the shorter R-arc $(r, x - 1, u, v - 1)$. This case corresponds to extending a rule match without error. For example, the distance on the R-arc $(r_x, 2, 1, 3)$ in figure 3.2 was computed from the distance on the R-arc $(r_x, 1, 1, 2)$ by matching the second word on the RHS of r_x ("cows") against the T-arc spanning nodes 2 and 3.

3. If $x > 0$ then let the distance on (r, x, u, v) be one greater than the distance on the R-arc $(r, x - 1, u, v)$. This case corresponds to inserting an extra word in the input phrase in order to match the x-th symbol on the RHS of the production rule. The distance is increased by one to reflect the insertion.

4. If $u < v$ then let the distance on (r, x, u, v) be one greater than the distance on the R-arc $(r, x, u, v - 1)$. This case corresponds to deleting a word from the input phrase, specifically the word on the T-arc which spans the nodes $v - 1$ to v. The distance is increased by one to reflect the deletion.

5. If more than one of the above steps is applicable, chose the one which gives the minimum distance.

An example is now given to demonstrate how the distance on an R-arc is computed. Let the rule of the R-arc be r_z as follows:

$$\text{S} \rightarrow \text{cows eat the grass}$$

By inspection, we see that the distance on an R-arc $(r_z, 4, 1, 5)$ on the input graph of figure 3.1 should be two – one for the deletion of the "all" arc between nodes 1 and 2 and one for the insertion of "the" as the third word in the production rule. A step-by-step derivation of this distance is shown in table 3.1. The first column of table 3.1 names a sub-arc whose distance is used to establish the distance of $(r_z, 4, 1, 5)$. The second column is the distance for the sub-arc. The right-most column of the table justifies the distance by referring to the case of specification 3.1 by which the distance was computed. Table 3.1 is not a complete proof of the distance for the R-arc $(r_z, 4, 1, 5)$ because it fails to enumerate all possible R-arcs for the production rule r_z, and such an enumeration is necessary due to case 5 of specification 3.1. Nor is table 3.1 an algorithm. Table 3.1 is only an example intended to show how the distance of any R-arc depends on the distances of smaller R-arcs.

Specification 3.1 is deficient in an important detail – it fails to mention what should be done if a nonterminal symbol is found on the RHS of the production rule r. The changes needed to support nonterminals in R-arcs are intuitively straightforward. When a nonterminal is encountered, one need only search the input graph for an NT-arc instead of a T-arc, and then add in the distance of the NT-arc when matching. Specification 3.2 will make this idea more precise.

R-arc	Distance	Justification
$(r_z, 0, 1, 1)$	0	case 1
$(r_z, 0, 1, 2)$	1	case 4
$(r_z, 1, 1, 3)$	1	case 2
$(r_z, 2, 1, 4)$	1	case 2
$(r_z, 3, 1, 4)$	2	case 3
$(r_z, 4, 1, 5)$	2	case 2

Table 3.1: Derivation of distance for the R-arc $(r_z, 4, 1, 5)$.

Specification 3.2 *Changes to specification 3.1 to support the presence of nonterminal symbols on the RHS of production rules.*

1. The basis case is unchanged.
2. When matching a nonterminal instead of a terminal, an NT-arc is used instead of a T-arc. But NT-arcs, unlike T-arcs, might span multiple nodes of the input graph, and more than one NT-arc might terminate at the node v. Hence, this case should search for all NT-arcs of the form $(\aleph, z, v)$ where $\aleph$ is the x-th RHS symbol of the production rule r and where $z \neq v$. (The computation of NT-arcs is described in section 3.1.4.) The distance on the R-arc (r, x, u, v) should be the sum of the distances on the NT-arc and on the R-arc $(r, x-1, u, z)$. The node z is chosen so that the distance is minimized.
3. Inserting a nonterminal is just like inserting a terminal, except that the added distance is not one but the length of the shortest string which can be generated from the inserted nonterminal. If $\aleph$ is a nonterminal, then call the length of the shortest terminal string which can be derived from $\aleph$ the *insertion cost* of $\aleph$. Note that the insertion cost for nonterminal $\aleph$ is also the distance on the NT-arc $(\aleph, v, v)$ for $1 \leq v \leq V$. If $\aleph$ is the x-th RHS symbol of the production rule r, then the distance on the R-arc (r, x, u, v) is the distance of the R-arc $(r, x-1, u, v)$ plus the insertion cost of $\aleph$.
4. There is no deletion case for nonterminals. Instead, a nonterminal may be deleted by using multiple terminal deletions as described in specification 3.1
5. As before, when more than one of the above cases apply, choose the case which gives the least distance.

3.1.4 Computation of Distance for NT-arcs

Compared to R-arcs, the computation of distances on NT-arcs is straightforward. Suppose $\aleph$ is a nonterminal and that all production rules of the CFG which contain $\aleph$ on their LHS are $\{r_1, r_2, \ldots, r_q\}$. Then, the distance on the NT-arc $(\aleph, u, v)$ is just the least of the distances on the R-arcs:

$$\begin{array}{c} (r_1, |r_1|, u, v) \\ (r_2, |r_2|, u, v) \\ \vdots \\ (r_q, |r_q|, u, v) \end{array}$$

Thus, the distance of a nonterminal is the distance found by completely matching any of its production rules.

3.1.5 Construction of the Parsing Graph

From the above descriptions it is seen that the distance with which the root arc is labeled depends upon the distances of several other NT-, R-, and T-arcs. The other NT- and R-arcs in turn depend upon still other NT-, R-, and T-arcs, and so forth until everything ultimately depends on the presence of T-arcs. The dependencies between arcs are made explicit by the parsing graph, which is described in detail in this subsection. The subsequent subsections will show how the distance on the root arc can be computed by a depth-first search of the parsing graph beginning at the root arc.

Nodes of the parsing graph come in three varieties. Each node of the parsing graph is labeled by a distance (just as each arc of the input graph is) and the different varieties of nodes are intended to show the different ways in which the distances on nodes may be computed. The allowed types of parsing graph nodes are:

1. *Leaf nodes* are nodes of the parsing graph which have no leaving edges. Every leaf node has a distance of 0.

2. *Minimization nodes* are nodes in the parsing graph for which the distance is computed by taking the minimum of the distances of all the node's children.

3. *Summation nodes* are nodes of the parsing graph for which the distance is the sum of all of the node's children.

In addition to distance labels on nodes, edges in the parsing graph may also be labeled by a distance when the parsing graph is first constructed. The edges cause a fixed increase in the computed distance when they are transversed. For example, suppose that A and B are two nodes of the parsing graph and that there is an edge from B to A. (Thus A is a child of B.) If A has a distance of D_A and the edge has a distance of δ, then the distance of A as seen by node B is really $A_D + \delta$.

The construction of the parsing graph from the arcs of the input graph is as follows: T-arcs correspond to leaves of the parsing graph. For each NT-arc in the input graph there is a single minimization node of the parsing graph. Call such a parsing graph node an NT-node. The children of this NT-node are the subgraphs (described in the next paragraph) which correspond to R-arcs upon which the NT-arc depends. Hence the distance for the NT-node will be the minimum of the distances on the child R-nodes, exactly modeling the distance computation for the NT-arc as specified in section 3.1.4.

The construction of parsing graph elements which correspond to R-arcs is more difficult, reflecting the increased complication associated with computing the distance of an R-arc. In the first definition of the parsing graph (just prior to the beginning of section 3.1.2) it was implied that there is a one-to-one correspondence between arcs of the input graph and nodes of the parsing graph. This is true for NT- and T-arcs, but for reason of simplicity in the algorithms to be described subsequently, it is useful for R-arcs of the input graph to sometimes have more than one corresponding node in the parsing graph. Consider that each R-arc corresponds to a small subgraph of the parsing graph, and call each such subgraph an *R-subgraph*.

The root of every R-subgraph is a minimization node. Suppose the corresponding R-arc is (r, x, u, v) and the x-th symbol on the RHS of production rule r is $\aleph$. The specification 3.3 tells

what leaving edges and what additional nodes must be added to the root node in order to complete this R-subgraph.

Specification 3.3 *The construction of the R-subgraph* (r, x, u, v) *given the existence of its root node.*

1. If $x = 0$ and $u = v$, then the root of the R-subgraph is a leaf node with a distance of 0. This case corresponds to case 1 of specifications 3.1 and 3.2. If this case applies, then all of the following cases should be disregarded.

2. If $\aleph$ is a terminal symbol and there exists a T-arc $(\aleph, v-1, v)$, then add a summation node, X, to the R-subgraph, make an arc from the root of the subgraph to X, and make arcs from X to both the T-arc $(\aleph, v-1, v)$ and the root of the R-subgraph for $(r, x-1, u, v-1)$. This case constructs the parsing graph so as to compute the distance according to case 2 of specification 3.1.

3. If $\aleph$ is a nonterminal symbol, then add to the R-subgraph V summation nodes $\{X_1, X_2, \ldots, X_V\}$ (recall that V is the number of nodes in the input graph) and make an edge from the root of the subgraph to each of these summation nodes. From each summation node, X_k, make edges which leave the subgraph and connect to the NT-node $(\aleph, k, v)$ and the root of the R-subgraph $(r, x-1, u, k)$. The primary focus of this case is the implementation of case 2 of specification 3.2. However, the summation node X_v is used to implement case 3 of specification 3.2. This is because the distance on the NT-arc $(\aleph, v, v)$ is always just the insertion cost of the nonterminal $\aleph$.

4. If $\aleph$ is a terminal symbol and $u < v$, then make an edge from the root of the subgraph to the root of the R-subgraph for $(r, x, u, v-1)$. Label the arc with a distance of 1. This case implements case 5 of specification 3.1.

5. Finally, if $\aleph$ is a terminal symbol and $x > 1$, then make an edge from the root of the subgraph to the root of the R-subgraph for $(r, x-1, u, v)$. Label the arc with a distance of 1. This case implements case 4 of specification 3.1.

An example R-subgraph

A simple example will help to elucidate the construction of an R-subgraph. Suppose the input graph is as shown in figure 3.1. Let the grammar be as follows:

S $\rightarrow$ all cows VP
VP $\rightarrow$ eat grass

This is a rather oversimplified grammar, but it will suffice for this demonstration. To the input graph of figure 3.1, add 7 R-arcs designated A–G and 4 NT-arcs designated H–K.

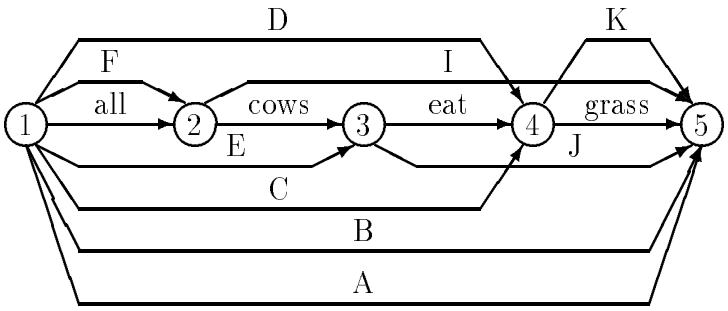

Figure 3.3: The input graph of figure 3.1 augmented with R-arcs and NT-arcs.

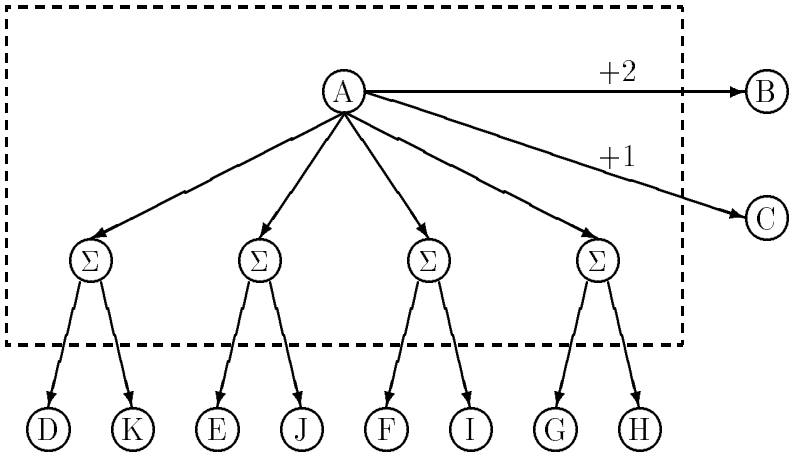

Figure 3.4: The parsing graph which corresponds to the input graph of figure 3.3.

A: ("S → all cows VP", 3, 1, 5)
B: ("S → all cows VP", 2, 1, 5)
C: ("S → all cows VP", 3, 1, 4)
D: ("S → all cows VP", 2, 1, 4)
E: ("S → all cows VP", 2, 1, 3)
F: ("S → all cows VP", 2, 1, 2)
G: ("S → all cows VP", 2, 1, 1)

H: ("VP", 1, 5)
I: ("VP", 2, 5)
J: ("VP", 3, 5)
K: ("VP", 4, 5)

The input graph with the addition of some of the R- and NT-arcs is shown in figure 3.3. (Arcs G and H are omitted from this figure for clarity.) Now consider the R-subgraph of the parsing graph which corresponds to the arc A in figure 3.3. A diagram of this R-subgraph is shown in figure 3.4. Only those nodes within the dashed box of figure 3.4 are properly a part of the R-subgraph. The other nodes in the figure show how the R-subgraph is connected to the outside. The arc of the parsing

graph from node A to node B corresponds to the insertion of the nonterminal VP in accordance with step 3 of specification 3.2. Similarly, the arc from node A to node C corresponds to the deletion of the T-arc between nodes 4 and 5 in accordance with step 4 of specification 3.1. The R-subgraph contains 4 summation nodes (each labeled by the symbol "Σ") which each correspond to the matching of the nonterminal VP against an NT-node, as described by step 2 of specification 3.2.

3.1.6 Walking the Parsing Graph

The distance of the root of the parsing graph may be efficiently computed by a depth first search. The depth first search of a cycle-free parsing graph is described first, since this case is much simpler and therefore easier to understand. However, most practical natural language grammars will generate parsing graphs that contain cycles. Hence a procedure for a depth-first walk of a parsing graph with cycles is subsequently described.

Cycle-free Parsing Graphs

Every node of the parsing graph is initially labeled with two values:

- An upper bound, named $\top$, on the distance for the node. The minimum distance ultimately computed must be less than or equal to this value. Initialize this value to $+\infty$.
- A lower bound, named $-$, on the minimum distance of the node. The minimum distance ultimately computed must be greater than or equal to this value. This value is initialized to zero.

Let the true minimum distance of the node be called D. Then by definition we have $- \leq D \leq \top$. The objective of the algorithm is to lower $\top$ and raise $-$ until they converge on a single value, and thus determine D.

Algorithm 3.1 *A recursive procedure for computing the distance of a node in a parsing graph when the parsing graph contains no cycles.*

Input:

1. F, a cycle-free parsing graph with every node initialized as described above.
2. A, a node in F, whose distance is to be computed.
3. ℓ, the limiting value. The calling function cannot use this node if its distance is greater than ℓ, so computation can be abandoned without consequence if $-$ exceeds ℓ. The outer-most call to this algorithm has $\ell = +\infty$.

Output:

1. $\top$, the upper bound estimate of node A's distance.

Method:

1. If A is a leaf in the graph, then return $\top = 0$.
2. If $\top = -$, then the exact distance has already been computed by a prior call to this procedure. Just return $\top$ without further computation.
3. If $\ell < -$, then additional computation will be of no benefit. Just return $\top$ without further computation.

4. If $- < \top < \ell$, then set $\ell \leftarrow \top$.
5. If A is a minimization node, then for each child node of A do the following:
   (a) Call algorithm 3.1 recursively with the given child and with a limiting value of ℓ. Let the resulting upper bound be x.
   (b) If $x < \top$, then set $\top \leftarrow x$.
   (c) If $\top < \ell$, then set $\ell \leftarrow \top$.
6. Otherwise, if A is a summation node with children g and h, then do:
   (a) Call algorithm 3.1 recursively on g and with a limiting value of ℓ. Let the resulting upper bound be x.
   (b) If $0 \leq \ell - x$, then call algorithm 3.1 recursively on h and with a limiting value of $\ell - x$. Let the resulting upper bound be y.
   (c) If $x + y < \top$, then set $\top \leftarrow x + y$.
7. If $\top \leq \ell$, then set $- \leftarrow \top$.
8. Return the value in $\top$.

Parsing Graphs With Cycles

Practical natural language grammars will usually result in parsing graphs which contain cycles. This is because natural languages are infinite languages, and therefore must be described by grammars which allow a nonterminal to derive a string which contains itself, or in other words, grammars which allow derivations of the form

$$A \rightarrow^{+} \alpha A \beta \tag{3.1}$$

where A is a nonterminal and α and β are strings of zero or more terminals and nonterminals. Let X be a node of the parsing graph which is labeled by the nonterminal A. Because of the way in which the parsing graph is constructed, there is a path from X to every node of the parsing graph upon which the distance of X depends, and so there must be paths from X to nodes corresponding to each symbol on the right-hand side of derivation 3.1. Now if all of the symbols in α and β are inserted into the input phrase (that is to say that step 3 of specification 3.2 is applied to every symbol in α and β), then any parsing graph node which corresponds to the left-hand side A of derivation 3.1 must also correspond to the right-hand side A in that derivation since the left- and right-hand sides of any derivation must span the same set of input symbols and α and β do not span any input symbols. X is such a parsing graph node, so there will be a path in the parsing graph from X back to itself – a cycle.

A cycle in a parsing graph is a set of strongly connected nodes. The entry point of a cycle is the node in the strongly connected set which the graph walking algorithm attempts to evaluate first. Once the algorithm enters a cycle, it can potentially begin circling around nodes in the cycle forever. This is not necessarily the case, since the limiter value, ℓ, will often prevent the full cycle from being explored, but the potential for the infinite loop does exist and must be addressed.

The infinite loop is broken at the first node in the cycle which the algorithm encounters. Every node in the parsing graph is assigned a marker with possible values BUSY and IDLE to indicate whether or not computation of the distance of that node is currently in progress. If the algorithm encounters a node marked BUSY, it knows it is in a cycle and can take appropriate action.

When a BUSY node (call it A) is seen, the algorithm is unable to compute the correct $\top$ value for that node, and must instead return whatever $\top$ value has already been computed. Usually the value returned is $+\infty$, though sometimes it may be smaller if children of A which are not a part of the cycle have already been explored. In either case, however, the value returned is not necessarily

the correct minimum distance for A. This does not damage the original computation of A (the computation of A which was started when A was encountered the first time), since the distance of node A can never be less than itself, and since the intervening nodes in the cycle cannot reduce the apparent distance of A but can only increase it or leave it unchanged. However, the incorrect distance returned at the second encounter of A can cause nonminimal distances to be computed for other nodes in the cycle. If the cycle is later reentered at a different point, then nodes in the cycle between the new entry point and A must be recomputed.

The crux of the problem is on step 7 of algorithm 3.1. This step in effect says for the node being computed: "I have tried every possible path out of this node, and the best distance I could find was $\top$, so the minimum distance, $-$, must be no less than $\top$." This statement is correct in a cycle-free parsing graph, but if the node being computed is an intermediate node in a cycle then this statement is false. The presupposition "I have tried every possible path" fails since the breaking of the cycle prevented exploration of at least one path. The conclusion, that $-$ must be no less than $\top$, is therefore invalid. As a result, the algorithm 3.1 must be modified so as not to perform step 7 if the node being computed is an interior node of a cycle.

The method for detecting whether or not a node is interior to a cycle involves the use of a global *blocking count*, which is the number of times that a cycle has been detected and broken. There is also a blocking count associated with each node of the parsing graph, which can be compared to the global blocking count to determine if the node is interior to a cycle. This per node blocking count is initially zero for every node in the parsing graph. The method for walking a parsing graph with cycles is shown in its entirety in algorithm 3.2.

Algorithm 3.2 *A recursive procedure for computing the distance of a node in a parsing graph when the parsing graph does contain cycles.*

Input:

1. F, a parsing graph with every node initialized as described above.
2. A, a node in F whose distance is to be computed.
3. ℓ, the limiting value. The calling function cannot use this node if its distance is greater than ℓ, so computation can be abandoned without consequence if $-$ exceeds ℓ. The outer-most call to this algorithm has $\ell = +\infty$.

Output:

1. $\top$, the upper bound estimate of node A's distance.

Method:

1. If A is marked BUSY, then increment the blocking count on A and the global blocking count. Return $\top$ without further processing.
2. If A is a T-node (a leaf in the graph), then return $\top = 0$.
3. If $\top = -$, then the exact distance has been computed before. Just return $\top$ without further computation.
4. If $\ell < -$, then additional computation will be of no benefit. Just return $\top$ without further computation.
5. If $- < \top < \ell$, then set $\ell \leftarrow \top$.
6. Mark A as BUSY and set its blocking count to zero.

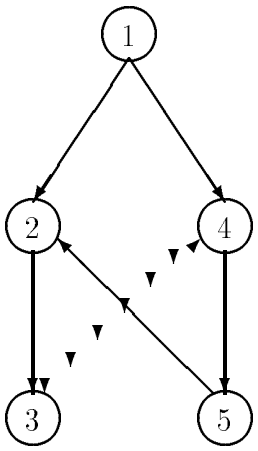

Figure 3.5: A small part of a hypothetical parsing graph which contains a cycle.

7. Record the global blocking count in a local variable *gbc*.
8. If A is a minimization node, then for each child node of A do the following:
   (a) Call algorithm 3.2 recursively with the given child and with a limiting value of ℓ. Let the resulting upper bound be x.
   (b) If $x < \top$, then set $\top \leftarrow x$.
   (c) If $\top < \ell$, then set $\ell \leftarrow \top$.
9. Otherwise, if A is a summation node with children g and h, then do:
   (a) Call algorithm 3.2 recursively on g and with a limiting value of ℓ. Let the resulting upper bound be x.
   (b) If $0 \leq \ell - x$, then call algorithm 3.2 recursively on h and with a limiting value of $\ell - x$. Let the resulting upper bound be y.
   (c) If $x + y < \top$, then set $\top \leftarrow x + y$.
10. Mark A as IDLE.
11. Reduce the global blocking count by the blocking count of A.
12. If $\top \leq \ell$ and the global blocking count equals *gbc* then set $- \leftarrow \top$.
13. Return the value in $\top$.

An example of recursion blocking in algorithm 3.2

The following example of the parsing graph walking algorithm is provided to help clear up any confusion the reader may have about how the algorithm works. Consider the portion of a parsing graph shown in figure 3.5. Each of the nodes in the parsing graph segment would, of course, have many arcs to other nodes of the parsing graph, but as these other arcs do not play a role in the example, they are omitted from the figure. Suppose that in the course of executing algorithm 3.2, node 1 of figure 3.5 is visited, and that the global blocking count is 326. While executing step 8, algorithm 3.2 calls itself recursively on node 2 of the graph. The local variable *gbc* is set to the global blocking count, the blocking count for node 2 is set to zero, and node 2 is marked BUSY. The algorithm is again called recursively for nodes 3, 4, and 5. At each of these recursions, then local *gbc*

variable is set to the global blocking count. Now the algorithm is called recursively again on node 2. But because node 2 has been marked BUSY, the condition on step 1 of algorithm 3.2 is statisfied. This causes the global blocking count, and node 2's blocking count to both be incremented, and the recursion to block. Execution now returns to node 5 and step 12 of algorithm 3.2. The condition fails because the *gbc* variable of node 5 is 326 but the global blocking count is 327. Consequently, the lower distance bound $-$ is not set equal to the upper distance bound $\top$ and the computation of distance for node 5 is left incomplete. The same thing happens at nodes 4 and 3. When the original execution of algorithm 3.2 for node 2 resumes, step 11 causes the global blocking count to be reduced back to 326. The test on step 12 now succeeds, and the computation of distance for node 2 is completed.

After node 2 is computed, the computation at node 1 resumes and the algorithm is called recursively for node 4. Node 4 was visited previously, during the computation of node 2, but the computation of node 4 was left incomplete because node 2 had blocked. Now the computation must be redone. The computation of node 4 causes node 5 to be recomputed as well. Algorithm 3.2 is called for node 2 again, as part of the computation of node 5, but step 3 causes the recursion to stop at this point. No further blocking occurs, so nodes 5 and 4 are successfully computed this time.

3.1.7 Correctness of MDP

This section will attempt to convince the skeptical reader that the distance computed by the MDP algorithm is the minimum distance between the input graph and the language described by the context-free grammar. Definitions of minimum distance for R-arcs and NT-arcs are given. Two lemmas and a theorem are then presented which show that the distance computed for each R-arc and NT-arc is the minimum distance for that arc. Because the final output of the MDP algorithm is the distance on the NT-arc labeled by the starting nonterminal and spanning the entire input graph and because all NT-arcs are labeled by the correct distance, then the output of the MDP algorithm must therefore be correct. Begin with the definitions.

Definition 3.1 *The Minimum Distance of an R-arc.*

Let G be a context-free grammar having initial nonterminal S which does not appear on the right-hand side of any production rule. Let the R-arc in question be (r, x, u, v) where r is a production rule in G and x is a nonnegative integer which is not greater than the number of symbols on the right-hand side of r and u and v are numbers of nodes in the input graph. Construct a new context-free grammar from G (call the new grammar G') as follows: remove from G every production rule what contains the initial nonterminal S, then add to G a single production rule which has S as the left-hand side and on the right-hand side has the first x symbols on the right-hand side of r. Let X be the set of all paths of T-arcs between u and v of the input graph. (Each such path is represented as an ordered list of words which are the labels on the T-arcs traversed.) Let Y the language of strings generated from G'. Let the function $D(a, b)$ be the minimum edit distance between two strings a and b. Then, the minimum distance of the R-arc (r, x, u, v) is defined to be

$$\min\{D(p, q) \mid p \in X \text{and} q \in Y\} \tag{3.2}$$

■

The definition of minimum distance for an NT-arc is very similar to that for an R-arc.

Definition 3.2 *The Minimum Distance of an NT-arc.*

Let the NT-arc in question be $(\aleph, u, v)$. The minimum distance of an NT-arc is defined exactly as the minimum distance of an R-arc except that the contruction of the new context-free grammar G' is changed slightly. G' is produced from G by first removing every production rule containing the initial nonterminal S and then adding a single production rule which as S as the left-hand side and for the right-hand side as the single nonterminal $\aleph$. ■

Lemma 3.1 *For a given R-arc H which has been added to the input graph, if the distance label of each arc upon which H directly depends according to specifications 3.1 and 3.2 is the minimum distance for that arc, and if the distance on arc H is computed according to specifications 3.1 and 3.2, then the distance on H is the minimum distance for H.*

Proof: Specification 3.2 is the same as specification 3.1 except that specification 3.2 assumes that the x-th symbol on the right-hand side of the production rule r in the R-arc (r, x, u, v) is a nonterminal. Because of this symmetry between specification 3.2 and specification 3.1 the proof for each specification is essentially the same. Therefore, for brevity's sake, only specification 3.2 is expounded upon in the sequel.

Call the x-th symbol on the right-hand side of production rule r by $\aleph$. If case 1 of specification 3.2 applys, then the sets X and Y both contain only the empty string and the distance between them is clearly zero. If case 1 does not apply, then the proof will be by contradiction. Suppose one of cases 2, 3, or 4 from specification 3.2 was used to compute the distance on H but the distance computed is not the minimum distance for H. Then there must be members of sets X and Y, call them p and q, such that the distance between those members, $D(p, q)$, is less than the computed distance for arc H. The left-most end of q must be composed of one or more symbols derived from $\aleph$. This is because q is derived from the right-hand side of r and the left-most symbol on the right-hand side of r is $\aleph$, (It is not possible for $\aleph$ to derive the empty string because the grammar was assumed to have no production rules with an empty right-hand side, with the possible exception of one production rule which has the initial nonterminal as its left-hand side. See section 3.1.2.) Call the left-most end of q which is derived from $\aleph$ by q_l and let the part of q which is not q_l be called q_r. Thus $q = q_r \cdot q_l$, where the dot means concatenation. When computing the distance between q and p, the tail q_l of q must be matched against some part of the left side of p. Call this segment p_l and call the rest of p by p_r. Thus $p = p_r \cdot p_l$. Because distance is additive and addition is associative, it must be the case that $D(p, q) = D(p_r, q_r) + D(p_l, q_l)$. But now observe that $D(p_r, q_r)$ is exactly the minimum distance on some R-arc $(r, x - 1, u, w)$ where w is a node in the input graph. (The value of w depends upon what symbols are in p_r.) Now consider the interpretation of $D(p_l, q_l)$. If p_l is the empty string, then $D(p_l, q_l)$ is the insertion cost of the string q_l. Because q_l is derived from $\aleph$ and the sum $D(p, q)$ is minimal, by supposition, then $D(p_l, q_l)$ must be the minimum insertion cost of the nonterminal $\aleph$. But if that where the case, then $D(p, q)$ would have been computed as the minimum distance for the arc H by case 3 of specification 3.2 which would violate the initial assumption. Hence p_l must be nonemtpy. Now if p_l is nonempty, then $D(p_l, q_l)$ is the minimum distance on some NT-arc $(\aleph, w, v)$, where w is the same node of the input graph mentioned above. (The distance must minimum because the sum $D(p, q)$ is minimal.) But if this is the case, then the distance $D(p, q)$ would have been computed as the minmum distance for X by case 2 of specification 3.2 (or case 4 of specification 3.1). Thus the assumption that the distance computed for X is not the minimum distance leads to a contradiction. ■

The next result is the same as lemma 3.1 except that it applies to NT-arcs instead of R-arcs.

Lemma 3.2 *For a given NT-arc H which has been added to the input graph, if the distance label of each arc upon which H directly depends according to the description in section 3.1.4 is the minimum*

distance for that arc, and if the distance on arc H is computed according to section 3.1.4, then the distance on H is the minimum distance for H.

Proof by contradition. Suppose the conditions of lemma 3.2 are satisified but the value computed for the distance of arc $H = (\aleph, u, v)$ is not the minimum distance for H. Then there must be strings $p \in X$ and $q \in Y$ such that the distance $D(p,q)$ is less than the distance computed for H. Because q is in Y, it must have been derived from one of the production rules which has $\aleph$ as its left-hand side. Let this production rule be called r. The distance on the R-arc $(r, |r|, u, v)$ must be the $D(p,q)$ or else the preconditions of this lemma would have been violated. But this requires that the distance on H be no more than $D(p,q)$ which violates the assumption and proves the lemma. ■

Lemmas 3.1 and 3.2 are now used to construct the induction step in an inductive proof of the correctness of the MDP algorithm.

Theorem 3.3 *The MDP algorithm is correct.*

Let the "depth" of a node in the parsing graph be the maximum number of unique arcs which can be traversed before a leaf node is reached. (For just this theorem, consider an R-subgraph to be a complete node, and let the distance on that node be the distance on the root of the R-subgraph.) By "unique arcs" it is meant that an arc may only be traversed once. Where this restriction not in place, many nodes would have an unbounded depth due to cycles in the parsing graph. Let $S(n)$ mean that the distance on nodes which have a depth of n is the minimum distance. The basis, $S(0)$, is true since the leaves are by definition labeled by their minimum distances. Lemmas 3.1 and 3.2 allow one to infer that if $S(0), S(1), \ldots, S(n-1)$ are all true then $S(n)$ must be true as well. This is because if an node X has depth n and depends upon node Y, then either the depth of node Y is $n-1$ or less, or else Y also depends (perhaps indirectly) on X. If Y has a lessor depth than X, then the induction hypothesis is satisfied for Y and it must therefore be labeled by the minimum distance. If Y depends on X, then the paths out of Y can be divided into two groups. $Y^{\downarrow}$ is the node Y with only those arcs which do not lead back to X and $Y^{\uparrow}$ is Y with only those arcs which do lead back to X. We have $Y = Y^{\downarrow} \cup Y^{\uparrow}$. Because each node effectively takes the minimum value of the nodes it depends upon (perhaps with some additional penalty distance – see specifications 3.1 and 3.2) the distance label on node Y will be the minimum of the two components $Y^{\downarrow}$ and $Y^{\uparrow}$. The distance of $Y^{\downarrow}$ is the minimal because the depth of $Y^{\downarrow}$ must be less than n. The depth of $Y^{\uparrow}$ is greater than n, however, and so the induction hypothesis does not guarantee that its distance is minimal. But because $Y^{\uparrow}$ depends on X and because distances can only increase as arcs are traversed, $Y^{\uparrow}$ can not contribute to the minimum distance of X and so it does not matter to the induction step if $Y^{\uparrow}$ is labeled by the minimum distance or not. From the above argument, one is able to conclude that $S(n)$ if $S(0)$ and $S(1)$ and ... and $S(n-1)$. Hence by total induction, $S(n)$ is true for all n. ■

3.1.8 Complexity of MDP

The worst case running time for computing the distance of a parsing graph using algorithm 3.2 is $O(V^3k^2)$ where V is the number of nodes in the input graph and k is the size of the grammar. Consider: the total number of nodes in the parsing graph is limited by the number of nodes in the input graph and by the size of the grammar to be $O(V^2k)$. The largest loop in the parsing graph can be no greater than $O(k)$, so algorithm 3.2 is called at most $O(k)$ times for each node in the parsing graph. The running time of each invocation of algorithm 3.2 is at most $O(V)$, by the loop at step 8. The worst case running time for the overall algorithm is obtained by multiplying these three factors.

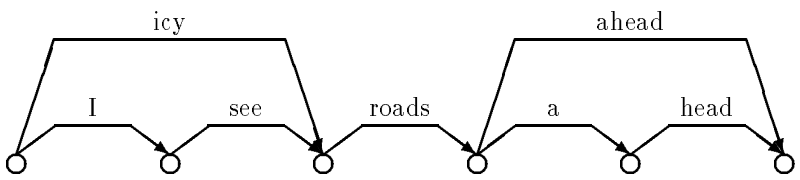

Figure 3.6: An typical input lattice to the MDP algorithm in which T-arcs are allows to span nonadjacent nodes of the input graph.

3.1.9 Extending MDP to Accept a Lattice as Input

In prior discussion of MDP, it has been assumed that the T-arcs always span adjacent nodes of the input graph. (An equivalent assumption has also been made by all prior descriptions of other minimum-distance parsing algorithms by other authors [2] [26] [27].) But this assumption is not necessary to the algorithm. It would work perfectly well to treat T-arcs as if they were NT-arcs, in which case they are allowed to span multiple nodes of the input graph. The result is a lattice input to the MDP algorithm instead of a simple input phrase. Figure 3.6 shows an example. And since the T-arcs are now being treated as if they were NT-arcs, there is no longer any reason to give every T-arc a distance of 0. It is acceptable, for example, to give each T-arc a distance which is based on the speech recognizer's confidence that the associated word was actually spoken by the user.

3.1.10 Extending MDP to use Non-uniform Insertion and Deletion Costs

This is also a good place to point out that the word insertion and deletion costs for the MDP algorithm need not be uniformly 1. A separate deletion cost can be associated with each T-arc in the input graph. Perhaps this deletion cost can be related to the speech recognizer's estimate of the likelihood that the word was actually spoken. This deletion cost can then be used in place of the constant 1 in case 4 of specification 3.1 and in the corresponding case 5 of specification 3.3.

Insertion costs also need not be 1. Insertion costs are not associated with T-arcs, however, but with symbols on the RHS of production rules. Thus the grammar for the MDP algorithm is used to specify nonuniform insertion costs to replace the constant 1 in case 3 of specification 3.1 and in the corresponding case 6 of specification 3.3. The use of nonuniform insertion costs creates a number of complications in definitions 3.1 and 3.2, specifically in the distance function $D(\cdot,\cdot)$. In definition 3.1, it is now necessary to store the insertion cost of each symbol in each string of the language Y computed from G' so that the distance function $D(\cdot,\cdot)$ will know what insert cost to use when computing the distance between strings. This change does not change the results of section 3.1.7, nor does it make the algorithm much more difficult to implement.

Section 6.1.2 describes how the nonuniform insertion costs are specified in the implementation. Special notations exist to cause a terminal symbol in the grammar to have a effectively infinite insertion cost, and thus require that the symbol be present in the input for the parser to be successful. A similar notation exists which causes a terminal symbol in the grammar to have a zero insertion cost, thus making that symbol optional.

3.1.11 Pruning the parsing graph

Algorithm 3.2 is an efficient method for finding the distance of the root of a parsing graph. However, a few minor changes can make the program run much faster in many cases.

Consider the step 4 of algorithm 3.2. This step causes the depth-first search to look no deeper into the parsing graph if the lower distance bound of the current node is greater than the maximum distance value that can be useful in the search. The effectiveness of step 4 at pruning the parsing graph can be enhanced by either increasing the lower distance bound, $-$, or by decreasing the limiting value, ℓ. Techniques for accomplishing both of these goals are described in the next subsections.

Increasing the lower distance bound $-$

The lower distance bound for every node in the parsing graph is initialized to zero. This bound may be increased as a result of computations on the interior nodes of a cycle, but that is unusual. Most often the value of $-$ at step 4 of algorithm 3.2 will be zero.

A larger lower distance bound can be estimated for many parsing graph nodes using algorithm 3.3.

Algorithm 3.3 *An algorithm for computing a quick estimate of the lower distance bound on a parsing graph node.*

Input:

1. A, a graph node whose lower distance bound is to be estimated.

Output:

1. $-$, the lower distance bound estimate.

Method:

1. If the lower bound of A has already been estimated by a prior call to this function, then return the previously computed estimate without further computation.
2. Initialize $-$ to zero.
3. If A is associated with an NT-arc, then call this algorithm recursively to estimate the distance of each child of A. Set $-$ equal to the minimum of these estimated distances.
4. If A is associated with the R-arc (r, x, u, v), then for every $0 \leq i < x$ do the following:
   (a) Let $\aleph$ be the i-th RHS element of production rule r.
   (b) If $\aleph$ is a nonterminal, then find B which is the parsing graph node associated with the NT-arc $(\aleph, u, v)$ and call this algorithm recursively to estimate the lower distance bound for B. Added this estimate to $-$.
   (c) If $\aleph$ is a terminal and there is no path from node u to node v of the input graph which contains a T-arc for $\aleph$, then increase $-$ by the i-th insertion cost of production rule r.
   (d) If $\aleph$ is a terminal symbol which is on some path from u to v then leave $-$ unchanged.
5. Return $-$.

To test the effectiveness of algorithm 3.3 at reducing the total parsing time, a set of 27 utterances taken from the user experiments described in [35] were parsed both with and without the use of algorithm 3.3, and the time required to complete these parses were compared. (The parser used in these tests actually implemented MDSDT, not MDP. But because MDSDT is just a combination of MDP and SDT, the results of these tests should still be valid.) On a 25MHz SparcStation2 with 64 megabytes of memory, the time needed to parse the 27 utterances without the use of algorithm 3.3 was 55.7 second. Only 24.6 second were required when algorithm 3.3 was used, for a savings of 56%.

Decreasing the limiting value ℓ

The limiting value ℓ is reduced by steps 8c and step 9b of algorithm 3.2. Consider first step 8c. The effectiveness of this step can be enhanced if on the first time through the loop the child node chosen for examination is one which returns a very small $\top$. Of course, it is impossible to tell in advance which children of the node under consideration will have the smallest values of $\top$. However, a reasonable guess can be made by first estimating $-$ for every child node, then ordering the children so that those with the smallest $-$ (and thus the greatest potential for a small $\top$) are tried first. This node sorting heuristic can also be used in step 9. Simply select as g the child with the smallest $-$ with the hope that the $\top$ will also be small and thus provide a small limiter value for the computation of h.

To test the effectiveness of sorting nodes by $-$, the experiment used to evaluate algorithm 3.3 was repeated with node sorting enabled. The time needed to parse the 27 test utterances was 20.3 seconds. This is a 17% speed improvement over the case where nodes were not sorted by $-$ and a 64% speed improvement over the case where algorithm 3.3 was not used at all.

3.2 Syntax-directed Translation

Syntax-directed translation (SDT) is a means of associating every string of one context-free language (CFL) with some string of a second CFL. In the parser, the first CFL is a representation of the user's natural language and is the language specified by the CFG which is input to the MDP algorithm. The first CFL was called the input language in previous discussion. The second CFL is called the *output language* of the SDT. In the implementation, the second CFL is a description of a class of Prolog predicates which represent meaning. Hence, the implementation uses SDT to assign a meaning to every well-formed utterance in the user's natural language.

A SDT can be thought of as a rewriting of the parse tree (not the parsing graph) in which leaf nodes are inserted and deleted and non-leaf nodes are reordered. This section will begin with a formal description of SDTs and then conclude with some examples.

3.2.1 Notation

A syntax-directed translation grammar (SDTG) is the pentuple $(\mathcal{N}, \Sigma, \Pi, P, S)$ where $\mathcal{N}$ is a set of nonterminal symbols, Σ is a set of input terminal symbols, Π is a set of output terminal symbols, $P = \{r_1, r_2, \ldots, r_{|P|}\}$ is a set of production rules, and $S \in \mathcal{N}$ is a special nonterminal which is the start symbol of the grammar. Each SDTG production rule, r_i, is a quadruple (n, α, β, J) where $n \in \mathcal{N}$ is the left-hand side (LHS), $\alpha \in (\mathcal{N} \cup \Sigma)^*$ is the right-hand side (RHS), $\beta \in (\mathcal{N} \cup \Pi)^*$ is the semantics of the production rule, and J is a one-to-one mapping from every nonterminal symbol in β to an instance of the same nonterminal in α. Because every nonterminal in β is mapped and

because the mapping is one-to-one, the number of occurrences of a nonterminal in α must be at least as great as the number of occurrences of that same nonterminal in β. [1]

The syntax for describing the SDTG used in the implementation is explained in words and with examples in the appendix section 6.1.2. The reader may wish to refer to this section as an aid to understanding the formal definition of a SDTG given above.

3.2.2 A Theory of Syntax-directed Translations

Every SDTG contains two separate CFGs as subsets. The first CFG is called the *input grammar* and describes a context-free language (CFL) called the *input language*. The input grammar of SDTG $(\mathcal{N}, \Sigma, \Pi, P, S)$ is $(\mathcal{N}, \Sigma, P_{in}, S)$ where $P_{in} = \{(n, \alpha) \mid (n, \alpha, \beta, J) \in P\}$. Thus the input grammar of a SDT is the SDT without the output terminal set Π and without the semantics β and mapping function J on every production rule.

The second CFG of a SDT is the *output grammar* which specifies the *output language*. The output grammar is $(\mathcal{N}, \Pi, P_{out}, S)$ where $P_{out} = \{(n, \beta) \mid (n, \alpha, \beta, J) \in P\}$. Thus the output grammar is just the input grammar with the input terminal set Σ replaced by the output terminal set Π and with the RHS of each production rule α replaced by the semantics of each production rule β.

Every SDT describes a *translation language* which is a set of ordered pairs, called *translations*, and which is a subset of $\{(x, y) \mid x \in L(G_{in}) \text{ and } y \in L(G_{out})\}$, where $L(G_{in})$ and $L(G_{out})$ are the input and output languages, respectively. Sometimes a translation is called a *complete translation* in order to distinguish it from a partial translation defined below. Not every translation (x, y) is in the translation language of an SDT, but only those translations which satisfy the mapping requirements set forth by the J mappings in the productions of the SDT. The following paragraphs will describe in more detail which translations are and are not in the translation language of a SDT.

A *partial translation* is a triple (x, y, M) where $x \in (\mathcal{N} \cup \Sigma)^*$ is a set of input terminals and nonterminals, $y \in (\mathcal{N} \cup \Pi)^*$ is a set of output terminals and nonterminals, and M is a one-to-one mapping from nonterminal symbols in y into nonterminal symbols in x. In any partial translation, the first string x is called the input string and the second string y is called the output string. Let the set of all partial translations for a grammar G be $\mathcal{T}_\mathcal{G}$. Define the relation $\Rightarrow_G$: $\mathcal{T}_\mathcal{G} \rightarrow \mathcal{T}_\mathcal{G}$ as carrying one partial translation of G into another according to the rules of G. (When it is clear from context which SDT is to be used, the G subscript on the $\Rightarrow$ operator will be omitted.) The relation $A \Rightarrow B$ is read as "B is derived from A" or as "A generates B".

The $\Rightarrow$ relation is intended to mimic the action of the relation with the same name in the theory of context-free languages. Thus, the relation

$$(x_1 \aleph x_2, y_1 \aleph y_2, M) \Rightarrow_G (x_1 \alpha x_2, y_1 \beta y_2, \hat{M})$$

holds if and only if

1. The mapping M specifies that the $\aleph$ in the output string of the left-hand partial translation corresponds to the $\aleph$ in the input string of that partial translation.

2. SDT G contains a production rule of the form $(\aleph, \alpha, \beta, J)$

3. The new mapping $\hat{M}$ is consistent with both the old mapping M and the production rule mapping J.

[1] [1] requires that the mapping from β to α be both one-to-one and unto. This thesis drops the onto requirement to obtain a slightly more useful and general result.

As in traditional formal language theory, $\Rightarrow^*$ is the transitive and reflexive closure of $\Rightarrow$.

Now define two special mappings, M_I and $M_\emptyset$. The *identity mapping* M_I is used in partial translations in which the order of nonterminals in the output string and in the input string is the same. M_I maps the k-th nonterminal in the output string into the k-th nonterminal of the input string. Hence, in the partial translation (S, S, M_I), the nonterminal S in the output string maps into the nonterminal S of the input string. The *null mapping* $M_\emptyset$ is used in partial translations which contain no nonterminals, and hence require no mapping. It is convenient to consider a complete translation as just a partial translation which contains no nonterminals and therefore uses the mapping $M_\emptyset$.

With this notation, it is now possible to precisely define a translation language. Let G be the SDT which is $(\mathcal{N}, \Sigma, \Pi, P, S)$. The translation language of G is the set of complete translations $\{(x, y) \mid (S, S, M_I) \Rightarrow_G^* (x, y, M_\emptyset)$. In words, the translation language of G consists of all translations which can be derived from the start symbol S of G using any number of applications of production rules from G. This is analogous to the way in which a context-free language is defined in terms of a context-free grammar.

3.2.3 A Pedagogical Example

The following example SDTG will translate a simple operator prefix expression into its equivalent operator postfix expression.

S → C S' S" : S' S" C
S → B S : S B
S → A : A
A → a : a
A → b : b
⋮
A → z : z
B → - : -
C → + : +
C → × : ×

In this SDTG, the terminal alphabet for both input and output languages is the lower-case letter which represent values and the operator symbols +, -, and ×. The nonterminals are spelled with upper-case letters and digits. The start symbol is S. The mapping from nonterminals in the semantics to nonterminals in the RHS of each production rule is indicated by the use of apostrophes following each nonterminal name. In the first production rule, for example, the first occurrence of S in the RHS corresponds to the first occurrence of S in the semantics, since both of these nonterminals are marked with a single apostrophe. In the second production rules, no apostrophe markings are required since no nonterminal occurs more than once in the semantics.

Now consider the following prefix expression

× + a - b c

In the more familiar infix notation, this expression would be

(a + -b) × c

The following table shows the sequence of partial translations which derives the prefix expression. The rule of the SDT which produced each step in the sequence is shown to the right of the partial translation.

S	:	S	Initial state
C S' S"	:	S' S" C	S → C S' S" : S' S" C
× S' S"	:	S' S" ×	C → × : ×
× S' A	:	S' A ×	S → A : A
× S' c	:	S' c ×	A → c : c
× C S' S" c	:	S' S" C c ×	S → C S' S" : S' S" C
× + S' S" c	:	S' S" + c ×	C → + : +
× + A S c	:	A S + c ×	S → A : A
× + a S c	:	a S + c ×	A → a : a
× + a B S c	:	a S B + c ×	S → B S : S B
× + a - S c	:	a S - + c ×	B → - : -
× + a - A c	:	a A - + c ×	S → A : A
× + a - b c	:	a b - + c ×	A → b : b

From this derivation one can see that the translation of the prefix expression is the postfix expression

a b - + c ×

Within the parser, translation from a string of words into a Prolog predicate occurs in an analogous fashion, though the translation is somewhat more complex due to the added complexity of both input and output languages.

3.2.4 A Practical Example

The following example derivation is presented in order to illustrate how SDT is used to convert English phrases into Prolog predicates within the Circuit Fix-it Shoppe program. The input phrase for this example is

no wire.

The correct translation of this input is the predicate

assertion(false,state(exist,wire(*,*),present))

There are 471 rules in the SDTG used for this example, but only 7 of these rules apply to the example input phrase. Symbols in the grammar which begin with an upper-case letter are nonterminals and symbols beginning with a lower-case letter are terminals. The nonterminal "StartState" is the grammar's start symbol. The relavant rules of the SDTG are these:

StartState → A' : A'
A → TIS NOT' WIRE' : assertion(NOT',state(exist,WIRE',present))
TIS →
WIRE → DET WIRE2' : WIRE2'
DET →
NOT → no : false
WIRE2 → wire : wire(*,*)

The sequence of partial translations is shown in the following table.

```
StartState  :  StartState
A           :  A
TIS NOT' WIRE'  :  assertion(NOT',state(exist,WIRE',present))
NOT' WIRE'  :  assertion(NOT',state(exist,WIRE',present))
no WIRE'    :  assertion(false,state(exist,WIRE',present))
no DET WIRE2'  :  assertion(false,state(exist,WIRE2',present))
no WIRE2'   :  assertion(false,state(exist,WIRE2',present))
no wire     :  assertion(false,state(exist,wire(*,*),present))
```

3.2.5 Limitations of SDT

The mapping from input to output languages in a SDTG is not arbitrary. Corresponding strings in the input and output languages of an SDTG must share a common set of nonterminals. Also, corresponding nonterminals in the input and output must have the same children, though the children may be ordered differently. These facts place a strong constraint on the kinds of translations that can be implemented with a SDTG. For instance, no one could reasonably argue for the existence of a SDTG which carried English into Japanese as these languages lack the necessary affinity of structure.

Yet, this weakness of SDT should not disqualify it from use as the agent for assigning meaning to sentences in a natural language. It only means that the underlying structure of the meanings and the original sentences be related. If one views the parser not as the ultimate translation engine, but rather as a device for bringing order and structure to a chaotic and error-filled input, as this thesis does, then the limitations of SDT do not present a serious hindrance.

3.3 Minimum-distance Syntax-directed Translation

This section will describe the algorithm for *minimum-distance syntax-directed translation* (MDSDT) which is, as its name implies, a combination of minimum-distance parsing and syntax-directed translation which were described in the preceding sections. The MDSDT algorithm is the algorithm actually used by the parser in the Circuit Fix-it Shoppe program mentioned in the introduction. The preceding discussions of MDP and SDT were only to motivate and help explain the material presented in this section.

Notable features of the MDSDT algorithm include:

1. It will accept a lattice as input.
2. It provides the N-best outputs.
3. It combines minimum-distance parsing and syntax-directed translation into a single algorithm.

3.3.1 N-best or N-ties outputs

The original implementation of MDSDT provided true N-best output. The N translations with the best scores (the smallest distances) became the output of the algorithm. However, after this implementation was used in a series of user experiments it was found that translations with a distance which is greater than the minimum distance could be ignored without reducing the overall accuracy of the parsing system. This observation led to a second implementation which will output

up to N translations which are all tied for minimum distance. If the first implementation is an N-best output, then call the second implementation an *N-ties* output.

N-ties output has a speed advantage over N-best output. With N-ties the parsing graph can be pruned more aggressively resulting in much less search to obtain the final answer. On the other hand, N-best provides a more general output. Though N-best was not found to yield better parsing accuracy in our experiments, future studies may find advantages to N-best output. For this reason, both N-best and N-ties algorithms will be presented.

3.3.2 An Overview of the MDSDT Algorithm

MDSDT may be viewed as MDP with some extra baggage. A thorough understanding of MDP is essential, then, if MDSDT is to be understood. An understanding of SDT, on the other hand, is very helpful but is not essential.

MDSDT uses all the same data structures as MDP – an input graph containing T-arcs, R-arcs, and NT-arcs and a parsing graph with nodes corresponding to arcs in the input graph. MDSDT data objects store more information, however. An MDP parsing graph node holds only the upper bound $\top$ and the lower bound $-$. In addition to these, MDSDT stores N strings drawn from $(\mathcal{N} \cup \Pi)^*$. These strings are called *incomplete hypotheses* if they contain nonterminals. If a string contains no nonterminal symbols, then it is called a *complete hypothesis*. A complete hypothesis is a string in the output language of the syntax-directed translation grammar (SDTG). The collection of all hypotheses on a node is the *hypothesis set*. If the N-best outputs are desired, then there must also be stored a distance with each hypothesis in the hypothesis set, but for N-ties output, the distance of each hypothesis is the same as $\top$ and need not be explicitly stored.

Modifications to MDP needed to create MDSDT are of two types.

1. MDP must be augmented to compute the hypotheses.
2. The rules for determining $\top$ and $-$ must be modified so that they will not shut off computation at a given node before all N of the hypotheses have been computed.

3.3.3 Operations on Hypothesis Sets

It is convenient to define several operations on hypothesis sets since this will make the description of the MDSDT algorithm more concise. The first operation will be $\top(H)$, read "*top* of H". The result of $\top(H)$ is a distance which will end up playing the same role as the variable $\top$ played in algorithm 3.2. Intuitively, $\top(H)$ is the maximum distance which an hypothesis may have if it is to be considered for inclusion in the hypothesis set H.

There are two versions of the $\top(H)$ function; one for use with N-best MDSDT and the other for use with N-tied.

Algorithm 3.4 *A procedure for computing $\top(H)$ for N-best MDSDT.*

Input:

1. An hypothesis set H.

Output:

1. A distance, D.

Method:

1. If the number of elements in H is less than N then set $D \leftarrow +\infty$.
2. If the number of elements in H equals N, then set D to the maximum distance over every hypothesis in H.
3. Return D.

Algorithm 3.5 *A procedure for computing $\top(H)$ for N-tied MDSDT.*

Input:

1. An hypothesis set H.

Output:

1. A distance, D.

Method:

1. If the number of elements in H is zero then set $D \leftarrow +\infty$.
2. If the number of elements in H is greater than zero then set D to the distance of any hypothesis in H.
3. Return D.

A second operation on hypothesis sets is *hypothesis union*, written $H_1 \cup H_2$. This binary operation takes all hypotheses of both operand hypothesis sets and combines them into a single hypothesis set which becomes the result. In the case of N-best computations, the resultant hypothesis set is truncated so that it holds no more than N hypotheses. When truncation occurs, hypotheses with the least distance values are retained. For N-tied computations, not only is the set truncated to N elements, but any hypothesis with a distance greater than the minimum distance is discarded as well.

The third and final operation which will be defined on hypothesis sets is *merge*, written $H_1 \otimes_{\aleph} H_2$. The merge operator takes a particular nonterminal symbol in every hypothesis of H_1, specifically by the subscript $\aleph$, and replaces it with every hypothesis in H_2.

Algorithm 3.6 *An algorithm for computing $H_1 \otimes_{\aleph} H_2$.*

Input:

1. Two hypothesis sets H_1 and H_2.
2. A designated nonterminal k in every hypothesis of H_1

Output:

1. A new hypothesis set H_{out}.

Method:

1. Initialize H_{out} to the empty set.
2. For every hypothesis in H_1 with distance d_1 and of the form $\alpha\aleph\beta$ where $\aleph$ is the designated nonterminal, do the following:
   (a) For every hypothesis in H_2 with distance d_2 and of the form γ, do the following:

i. Create a new hypothesis $h = \alpha\gamma\beta$ with a distance of $d_1 + d_2$.

ii. Add h to H_{out}, pruning the set as appropriate depending upon whether N-best or N-ties output is desired.

3. Return H_{out}.

The merge operator is both complex and intuitively obscure, so further discussion and an example are perhaps warranted. Realize that an incomplete hypothesis is simply an incompletely derived string in the output language of the STDG. In this paradigm, the merge operation on hypothesis sets is equivalent to the derivation operator ("$\Rightarrow_G$") from context-free grammar theory. In other words, the merge operation replaces a single nonterminal with a sequence of terminals and other nonterminals which are derived by the original nonterminal. Of course there are complications with multiple strings and mapping functions and whatnot, but the reader should try to avoid being distracted by these details. Consider an example in which the terminals are lower-case letters and nonterminals are capitals. Suppose the left operand, H_1, is the following:

```
aBcbAce
abccABe
abAeeB
```

The left operand of $\otimes$ is the counterpart to the left operand on the $\Rightarrow_G$ operator. The right operand of $\otimes$ is the counterpart to the production rule used for the derivation step. Suppose that H_2, the right operand, represents the nonterminal `A` and has the value

```
uvXwYz
vXY
YwwX
```

Under this circumstance, the hypothesis set for $H_1 \otimes_A H_2$ would be (assuming $N \geq 9$ and the hypothesis set is untruncated) the following:

```
aBcbuvXwYzce
aBcbvXYce
aBcbYwwXce
abccuvXwYzBe
abccvXYBe
abccYwwXBe
abuvXwYzeeB
abuXYeeB
abYwwXeeB
```

3.3.4 The MDSDT Algorithm

As with MDP, the MDSDT algorithm begins with the construction of a parsing graph from the input graph. After the parsing graph is constructed (perhaps implicitly), algorithm 3.7 is called to compute the hypothesis set of the root of the parsing graph. This hypothesis set becomes the output of the MDSDT algorithm.

Algorithm 3.7 shows the details. The same basic algorithm works for both the N-best and N-tied outputs depending upon which version of the $\top(H)$ function is used.

Algorithm 3.7 *A recursive procedure for computing the N-best or N-tied hypotheses for a node in a parsing graph.*

Input:

1. A, a graph node whose distance is to be computed.
2. ℓ, the limiting value. The calling function cannot make use of any hypothesis which has a distance greater than this value, so computation can be abandoned if it becomes known that no more hypotheses with distances less than or equal to this value will be found.

Output:

1. An hypothesis set, H associated with the node A.

Method:

1. If A is marked BUSY, then increment the blocking count on A and the global blocking count. Return H without further processing.
2. If A is marked COMPLETE, the return H.
3. Estimate $-$ using algorithm 3.3.
4. If $\ell < -$, then additional computation will be of no benefit. Just return H without further computation.
5. If $\top(H) < \ell$, then set $\ell \leftarrow \top(H)$.
6. Mark A as BUSY and set its blocking count to zero.
7. Record the global blocking count in a local variable gbc.
8. If A is a minimization node, then for each child node of A do the following:
   (a) Call algorithm 3.7 recursively with the given child and with a limiting value of ℓ. Let the resulting hypothesis set be $\hat{H}$.
   (b) Set $H \leftarrow H \cup \hat{H}$.
9. Otherwise, if A is a summation node with children g and h, then do:
   (a) Call algorithm 3.7 recursively on g and with a limiting value of ℓ. Let the resulting hypothesis set be x.
   (b) Call algorithm 3.7 recursively on h and with a limiting value of $\ell - B(x)$, where $B(x)$ is the minimum distance of every hypothesis in x. Let the resulting hypothesis be y.
   (c) Every summation node is the child of an R-node. Suppose the R-node for this summation node is associated with the R-arc (r, i, u, v). Then node h is also an R-node associated with $(r, i-1, u, w)$. The hypotheses in y should all be copies of the RHS of production rule r with some of the nonterminals expanded. Yet, the i-th symbol on the RHS of r should be a nonterminal which is still unexpanded in every hypothesis of y. For the purposes of the $\otimes$ operator in the following step, call this nonterminal $\aleph$.
   (d) Set $H \leftarrow H \cup (y \otimes_{\aleph} x)$.
10. Mark A as IDLE.
11. Reduce the global blocking count by the blocking count of A.
12. If $\top(H) \leq \ell$ and the global blocking count equals gbc then set $- \leftarrow \top(H)$ and mark A as COMPLETE.
13. Return H.

3.3.5 Complexity of MDSDT

The worst case running time for the MDSDT algorithm is computed in the same way that the worst case running time was computed for MDP, but with an extra term added to disclose the computation needed for processing the hypothesis sets. The hypothesis union operation at step 8b requires time $O(N \log N)$ assuming that the maximum number of hypotheses is N. The hypothesis merge in step 9d requires time $O(N^2 \log N)$. Combining these factors with the previously computed worst case running time for MDP yields $O(V^3k^2N \log N + V^3kN^2 \log N)$ as the worst case running time for MDSDT.

3.4 The Use of Dialog Expectation

The MDSDT algorithm described above is used by the parser to compute a set of up to N hypotheses of the meaning of the user's input. This section will describe how dialog expectation is used to select one of these N hypotheses to be the final output of the parser.

3.4.1 Computation of Dialog Expectation

Dialog expectation is a collection of hypotheses which describe what the dialog system anticipates the user to say next. Associated with each dialog expectation hypothesis is a distance which indicates how strongly that hypothesis is anticipated. A small distance marks a highly anticipated hypothesis, and a large distance is used to label an hypothesis which is considered to be an unlikely input.

Dialog expectation is computed by the dialog controller. See figure 1.1 on page 2. Since the dialog controller is outside of the dashed box in this figure, it does not come under the purview of this thesis and its operation will not be described in detail. Just as the word lattice generated by the speech recognizer is considered to be an input to the parser, and is accepted without question, so too the dialog expectation is just another input to the parser over which the parser has no control.

In all implementations of the parser described in this thesis, the dialog expectation was generated by the dialog controller described in [35]. This does not preclude the possibility of using a completely different algorithm for generating dialog expectation. Nevertheless, for the sake of completeness the algorithm used to compute the dialog expectation in the implementations will be briefly summarized.

The state of a dialog is viewed as a sequence of nested contexts, each more specific and focused than its predecessors. Meanings which relate to the innermost context and can be understood in that context are made into dialog expectation hypotheses with small distance values. Meanings which can only be interpreted in the first enclosing context generate dialog expectation hypotheses with large distance values. This progression continues, so that meanings which can only be understood in more remote contexts are used to generate dialog expectations with increasingly larger distances.

3.4.2 Wildcards Within Dialog Expectation Hypotheses

Within any given context, there are thousands of possible expectations. So that the dialog controller will not have to list this many hypotheses when generating the dialog expectation, *wildcards* are used to represent an unspecified subcomponent of the hypothesis. As an example, suppose the SLS says to the user "What is the voltage at connector 46?" Without the use of a wildcard, the dialog expectation following this utterance might be:

VoltageIs(00.0)
VoltageIs(00.1)
VoltageIs(00.2)
⋮
VoltageIs(99.9)

It makes much more sense in this example to express the dialog expectation as a single hypothesis with a wildcard:

VoltageIs(*)

This is perhaps an extreme example, though not an uncommon one since the SLS frequently needs to receive a numeric input. But, wildcards are used to hold a place for more than just numbers. For instance, the entire class of meanings equivalent to the question "Where is the *object*?" is usually encoded as a single expectation with a wildcard holding the place for the unspecified subject of the sentence.

In the implementations, the dialog controller was coded in Prolog, and so wildcards were easily and naturally encoded as uninstantiated variables.

3.4.3 Wildcards Within MDSDT Hypotheses

Wildcards may also appear in the hypotheses which are output by the MDSDT algorithm, though the MDSDT algorithm itself does not understand the significance of wildcards and treats them no differently than any other symbol in the output terminal alphabet Π. The grammar used by the MDSDT algorithm can be constructed so that ellipsis and anaphora in the user's input utterance are represented in the hypotheses by wildcards. For example, the input phrase "The switch is in the up position" may be translated into

assertion(true,position(switch,up))

whereas the phrase "It is in the up position" or even the elliptical utterance "in the up position" would be translated as

assertion(true,position(*,up))

with the wildcard symbol "*" used to signify that the subject of the sentence was left unspecified.

3.4.4 Matching MDSDT and Dialog Expectation Hypotheses

The output of the parser is obtained by matching an MDSDT hypothesis with a dialog expectation hypothesis. The matching is performed by expanding wildcards in the MDSDT hypothesis and the dialog expectation hypothesis or both so as to make the two hypotheses identical. As an example, suppose the dialog expectation contains an hypothesis of the form

assertion(true,position(switch,*))

which means "the switch is in the X position" where X is unknown. Such an expectation might result if the SLS had just asked the user "What is the position of the switch?" Let the output of the MDSDT algorithm in this example be

assertion(true,position(*,up))

which is a rendering of "it is up". The hypothesis which results from matching is

assertion(true,position(switch,up))

or "the switch is in the up position."

The process of matching MDSDT and dialog expectation hypotheses is very similar to *unification* in logic [30, pp. 142–143] and in the Prolog programming language. (In fact, Prolog unification was used to perform matching in the first implementation.) If each wildcard is considered a singleton variable then the processing of matching MDSDT and dialog expectation hypotheses is equivalent to finding the most general unification of those hypotheses.

3.4.5 Selecting the Best Hypothesis

When an hypothesis from the MDSDT algorithm is matched against a dialog expectation, the new hypothesis which results from the matching is assigned a distance which is based upon the distances of the hypotheses input to the matching. Let the distance of the MDSDT hypothesis be U, and let the dialog expectation distance be E. The distance on the hypothesis which results from matching is C

$$C = f(U, E) \tag{3.3}$$

where function f is called the *expectation function*. It is customary to refer to U as the *utterance cost*, to E as the *expectation cost*, and to C as the *total cost* of an hypothesis. The final output of the parser is the hypothesis which has been successfully matched with a dialog expectation and which has the smallest total cost, C. If two or more hypotheses have the same C, one is selected at random. If no hypotheses can be successfully matched with dialog expectation then the parse fails.

3.4.6 The Expectation Function

In the initial implementation of the parser, the expectation function was computed as a linear combination of utterance cost and expectation cost.

$$f_1(U, E) = \beta \cdot U + (1 - \beta) \cdot E \tag{3.4}$$

The parameter β in equation 3.4 varies between 0 and 1 to control the relative importance of utterance and expectation cost. When β is near 1, utterance cost is the most important factor in choosing the parser's final output, but when β is near 0 the expectation cost predominates. The actual value chosen for β will of course also depend on the exact way in which distances are computed and on the relative scales of utterance and expectation costs.

Experimental data from the first implementation of the parser (see section 3.5) indicated that the best results were obtained when β was close to 1. Due to this observation, the second implementation of the parser implemented a new cost function.

$$f_2(U, E) = \begin{cases} E & U = U_{min} \\ +\infty & \text{otherwise} \end{cases} \tag{3.5}$$

In equation 3.5, the factor U_{min} is minimum distance of any hypothesis output by the MDSDT algorithm. The effect of using this new expectation function was that expectation cost is now ignored expect to break ties between two hypotheses with the same utterance cost. (Two other expectation functions similar to equation 3.5 are described in section 3.5.5.)

Coincident with the incorporation of equation 3.5 as the expectation function, the MDSDT algorithm was converted from the N-best to the N-ties method of recording translations. The

N-ties method is faster, due to the greater opportunity for tree pruning, and with expectation function 3.5 there is no reason to remember hypotheses with distances greater than the minimum distance because such hypotheses will never become the final output of the parser. (This is not strictly true. A non-minimal hypothesis might become the parser's output if none of the hypotheses with minimum distance match any dialog expectation hypothesis. In practice, however, this never happened.)

3.5 Experimental Evaluation of the Parser

The first implementation of the parser was used in a series of experiments in which users with no special knowledge of computers conversed with the SLS to obtain help in repairing an electronic circuit. Data obtained from these experiments indicates that the the parser performed well. This section will briefly overview the experiments, describe how the data was analyzed, and summarize the results of the analysis.

3.5.1 Experimental Design

The SLS used in the experiments consisted of a commercial speech recognizer, the parser designed in this thesis, a natural language dialog system developed by Ronnie Smith [35] [36], and a commercial speech synthesizer. The speech recognizer had a vocabulary of 125 words and was operated without the use of a grammar for limiting search. Connected speech was used. The speech recognizer did not output a lattice of word possibilities, as one would hope, but only the single best guess of the words spoken. Additional word possibilities were added to this best guess outside of the speech recognizer by a preprocessor to the parser. The additional words added to the input graph were based on empirical observations of the kinds of recognition errors that the speech recognizer was prone to make.

3.5.2 Data Collection

During the experiments, the users spoke 2804 utterances to the SLS. Information from these utterances was collected and converted to a machine-readable format. The following information was collected:

- The sequence of words actually spoken by the user. These were manually entered by the experimenters based on the audio recordings of the experiment.
- The sequence of words recognized by the speech recognizer. This information was recorded automatically during the experiments.
- The set of hypotheses which resulted from matching the best 10 hypotheses computed by the MDSDT algorithm with dialog expectation hypotheses. An utterance cost and expectation cost for each hypothesis was also recorded.
- The final output of the parser.
- The text spoken by the dialog controller immediately prior to the user's utterance, and notes concerning the user's utterance which were entered by the person who transcribed the utterance from the audio tapes. This information was used to assist in manually judging the correctness of each parse.

After the above information was collected and audited to remove errors, the following additional features of each utterance were added manually.

- A notation was made on each record to indicate whether or not the output of the parser was the meaning intended by the user.
- A second notation was made on each record to indicate whether or not the parser would have found the user's intended meaning if the speech recognizer had made no errors.

The judgment of whether or not the parser found the user's intended meaning is often very subjective. For example, suppose the user says "the LED is alternately flashing a one and a seven", but due to misrecognition of one or more words of the utterance, the parser determines the meaning to be the equivalent of "the LED is displaying a one and a seven and the one is flashing". In this example (which occurred in the actual data), the information that the seven is flashing has been lost. Nevertheless, the parser clearly had the right idea about what the user meant, even though it was wrong in one minor detail. In a case such as this, does one judge the parse to be correct or incorrect? Uncertainty about the parser's correctness can arise due to reasons other than misrecognitions. What does one do, for example, if the user's intended meaning is not clear even to another person? Should the parser be faulted for failing where even a human expert is puzzled? Suppose, as another example, the user speaks a phrase whose meaning cannot possibly be expressed in the mathematical language of the dialog controller. Any attempt to parse such an utterance is doomed to fail even before it begins. Should the parser be faulted for failing to perform the impossible?

The following decision criteria were used to judge the correctness of a parsed utterance:

1. If the parser finds exactly the intended meaning, mark the parse as "correct".
2. If the meaning found by the parser matches most of the meaning of the spoken utterance, and nothing in the parser's result contradicts the user's intended meaning, then mark the parse "correct".
3. If the parser's output contains any information which contradicts information which the user intended to communicate, mark the parse as "wrong".
4. If the user's meaning cannot be represented in the dialog controller's language, then mark the parse "wrong".
5. If the parser's result and the spoken utterance do not contradict, but less than half of the meaning of the spoken utterance is preserved in the parser's output, then mark the parse as "wrong".
6. If the user's meaning is ambiguous, then mark the parse "correct" if the meaning found by the parser matches any possible interpretation of what the user said.

Data from the experiments was analyzed independently by the author of this thesis and by Ronnie Smith, the author of [35]. These analyses occurred without prior consultation as to how the analyses should be conducted or as to what methods of analyses should be used. The results of the two analyses are different, but only slightly so. The results agree within 1%. The data analysis by Smith can therefore be viewed as an independent confirmation of the analysis undertaken in this thesis.

89	88	88	79	**75**	72	68	65	63

Table 3.2: PWC

85	84	80	74	**67**	63	54	54	51

Table 3.3: Modified PWC

3.5.3 Performance of the Speech Recognizer

Neither the implementation nor the performance of the speech recognizer is properly a part of this thesis. Nevertheless, it seems good to report how well the speech recognizer was able to transcribe the users' speech because the accuracy of the speech recognizer plays an important role in determining the accuracy of the parser. This information may also play a role in estimating how well the parser presented in this thesis will perform using a different speech recognizer front end.

The first measure of the speech recognizers performance is called the percent of words correct, or "PWC". The PWC is the number of words spoken by the user which were correctly recognized by the speech recognizer divided by the total number of correct words and multiplied by 100.

$$\text{PWC} = 100 \cdot \frac{\text{number of correctly recognized words}}{\text{number of words actually spoken}} \tag{3.6}$$

When a spoken word is misrecognized, or is not recognized at all, the PWC is reduced. When extra words are inserted by the speech recognizer, on the other hand, the PWC remains unchanged.

The PWC for the commercial speech recognizer used in the experiments described above is reported in table 3.2 below. Each number in the normal font is the PWC for one of the eight experimental subjects. The large bold-face number is the PWC for all subjects combined. This manner of presentation is intended to give the reader an intuitive feel for how much the PWC varied among the experimental subjects. A similar presentation format will be used for statistics throughout this section.

A second measure of the speech recognizer's accuracy will be called modified PWC. As with ordinary PWC, modified PWC is computed by taking the number of words correctly recognized, multiplying by 100, and then dividing. The difference is that in ordinary PWC, the divisor is the number of words actually spoken, but in modified PWC the divisor is the number of words output by the speech recognizer. We have:

$$\text{Modified PWC} = 100 \cdot \frac{\text{number of correctly recognized words}}{\text{total number of words recognized}} \tag{3.7}$$

With modified PWC, insertion errors do make a difference, since they increase the divisor and thus lower the score. Another view of modified PWC is that it is the percentage of words seen by the parser which were actually spoken by the user. The modified PWC for the experiments is shown in table 3.3.

A third measure of speech recognition accuracy is called the error rate. The error rate is the number of errors in an utterance divided by the correct length of the utterance, and multiplied by 100.

$$\text{Error Rate} = 100 \cdot \frac{\text{number of errors}}{\text{number of spoken words}} \tag{3.8}$$

71	67	57	43	**41**	35	23	17	16

Table 3.4: Error Rate

3.3	2.8	1.8	**1.6**	1.3	1.0	0.9	0.9	0.7

Table 3.5: Percentage of utterances containing one or more words not in the speech recognizer's vocabulary

The number of errors in an utterance is the minimum number of insertions, deletions, and substitutions needed to transform what was spoken into what was output from the speech recognizer.[2] The error rate for the speech recognizer in the experiments is shown in table 3.4.

The speech recognizer could understand a vocabulary of only 125 words. Occasionally, the user will speak some word which is not in this vocabulary. The percent of utterances which contain one or more words which are not in the 125-word vocabulary is shown in table 3.5.

The fact that very few utterances contained any words which were not in the speech recognizer's vocabulary suggests that the limited vocabulary was not a serious detriment to the overall performance of the system.

The percentage of utterances which were recognized without error is reported by table 3.6. This value is a good estimate of an upper bound on the performance of a parser which cannot handle recognition errors. Trivial utterances (that is, utterances which consist of only a single word – typically "yes" or "no") and utterances which contain words not in the speech recognizer's vocabulary were not considered when computing numbers reported in table 3.6.

Comparison to other speech recognizers

Table 3.7 below shows error rates for several research speech recognition systems. The *perplexity* shown in this table is a measure of how many words the speech recognizer must choose from at any given moment. A larger perplexity tends to increase the error rate, since having more words to choose from increases the chance of choosing the wrong word. For purposes of comparison, table 3.7 also shows the error rate measured on the Verbex 6000 during the experiments, and the error rate of a human listener from an experiment described in the following subsection.

Comparison to humans

The following simple experiment was conducted in an effort to measure the speech recognition performance of humans under conditions similar to those experienced by mechanical speech recog-

[2] An efficient algorithm for computing the minimum number of errors is described in [37].

58	47	40	**23**	16	13	10	9	3

Table 3.6: Percentage of utterances which were recognized without error

System Name	*Perplexity*	*Error rate*
ARM [33]	497	13%
BBN SLS [7]	700	13%
SPHINX [25]	997	29%
SPICOS [32]	124	9%
SUMMIT [47]	1000	56%
Verbex 6000	124	41%
Human	122	3%

Table 3.7: Error rates of several speech recognizers

97	91	90	88	83	**83**	80	73	64

Table 3.8: Percentage of all inputs which were correctly parsed

nizers. A list of 100 random utterances was read by the experimenter to a volunteer subject over a telephone. The listener typed what he heard into a computer for later analysis. Each utterance had a length which was uniformly distributed between 2 and 7 words. The words were selected at random[3] from a 122-word vocabulary which was essentially the same as the 125 word vocabulary used by the speech recognizer. The utterances were randomized in order to prevent the hearer's expert knowledge of English grammar from assisting in his word recognition. The telephone was used for two reasons. First, the telephone blocked all non-verbal communication, such as facial expressions or hand gestures, which might have subconsciously aided in word recognition. Second, the telephone limited the speech signal bandwidth and thus prevented the hearer from using sound information which is not also normally available to mechanical speech recognizers. The style of speaking used in this experiment was connected speech.

Out of 462 words spoken in this experiment, there were 13 errors – 11 substitutions and 2 deletions. The PWC was 97.2, and the error rate was 2.8 percent.

3.5.4 Performance of the Parser

The percentage of all utterances for which the parser was able to find the correct meaning is shown in table 3.8.

Slightly more than half (52%) of the utterances are trivial one-word responses to questions from the computer. If these trivial utterances are removed from consideration, the the percentage of utterances for which the parser found the correct meaning is reduced to the values shown in table 3.9.

Most of the parsing errors were a result of speech recognition errors. If the text of what was actually spoken were input to the parser instead of the text which the speech recognizer produced, the percentage of all utterances which would have been parsed correctly increases to the values shown in table 3.10. The values in this table do not change significantly if only non-trivial utterances are considered.

[3]The `drand48` function in the C library on UNIX was used to select both the length of each utterance and the words contained within the utterance.

99	85	83	78	**69**	65	63	56	46

Table 3.9: Percentage of nontrivial utterances which were correctly parsed

100	99	99	98	98	98	**97**	96	89

Table 3.10: Percentage of all utterances that would have been correctly parsed assuming no speech recognition errors

It is interesting to consider what percentage of inputs to the system were ungrammatical.[4] In [9], Eastman and McLean report that about a third of the inputs to a natural language database query system were not syntactically well formed. On the other hand, Fineman argues in [10] that ungrammatical input is not normally a problem since only about 2% of natural language inputs are ungrammatical. Our experimental data supports the findings of Eastman and McLean. Even without speech recognition errors, the percentage of non-trivial utterances that were ungrammatical was more than a third. Table 3.11 shows the specific values. When speech recognition errors are considered, the percentage of ungrammatical inputs was much larger, as is shown in table 3.12 The counterpart to table 3.11 for all utterances, both trivial and non-trivial, is shown in table 3.13. This is the percentage of all utterances which would have been ungrammatical even if the speech recognizer had made no errors. Finally, table 3.14 shows the percentage of all utterances, both trivial and non-trivial, which are grammatically ill-formed after speech recognition.

In interpreting the previous results, the reader should remember that the the translation grammar for this SLS was developed with full knowledge that the parser would be able to interpret ungrammatical inputs, and so little effort was made to give the translation grammar complete language coverage. Nevertheless, the system would probably have never achieved a parsing accuracy as high as it did had it not been able to guess the meaning of highly ungrammatical inputs.

Figures 3.7, 3.8, and 3.9 are diagrams of the relationship between the number of non-trivial utterances which were parsed correctly and measures of the speech recognizer's accuracy. Each point in these diagrams represents the results from a single experimental subject. The ordinate of each point is the percent of correct parses and the abscissa is the speech recognizer accuracy. Let the variable C represent the percent of non-trivial utterances which were correctly parsed, let P be the PWC, M is the modified PWC, and let R be the speech recognizer's error rate. Then linear equations which predict the value of C with minimum squared error, given either P, M, or R are respectively equations 3.9, 3.10, and 3.11 below.

$$C = 1.15983P - 12.5233 \tag{3.9}$$

[4] An *ungrammatical* input is any input which is not in the input language of the translation grammar.

49	46	43	41	**37**	35	28	28	25

Table 3.11: Percentage of non-trivial utterances which would have been ungrammatical even without speech recognition errors

52	51	50	48	46	**43**	38	32	29

Table 3.12: Percentage of non-trivial utterances which were ungrammatical when speech recognition errors are considered

26	25	24	24	**21**	20	17	14	12

Table 3.13: Percentage of all utterances which would have been ungrammatical even without speech recognition errors

87	70	56	53	**52**	43	41	40	23

Table 3.14: Percentage of all utterances which were ungrammatical when speech recognition errors are considered

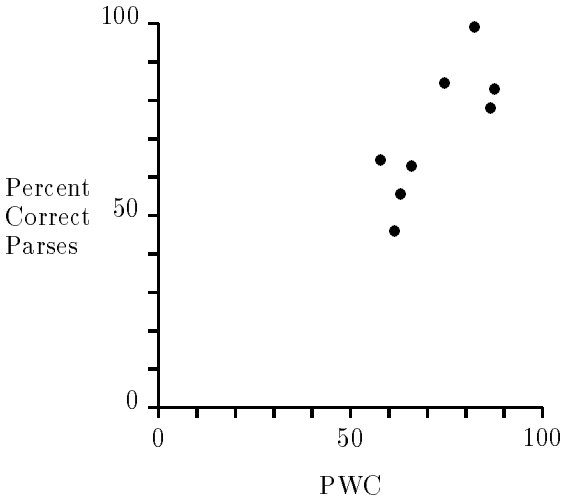

Figure 3.7: The parser accuracy on non-trivial utterances plotted against the percent of correctly recognized words for each experimental subject.

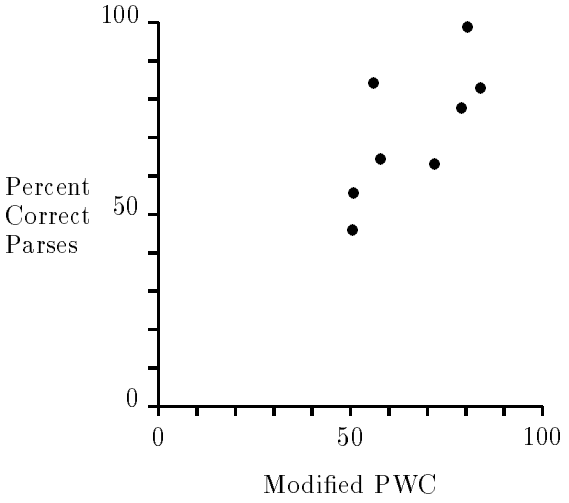

Figure 3.8: The parser accuracy on non-trivial utterances plotted against the *modified* PWC for each experimental subject.

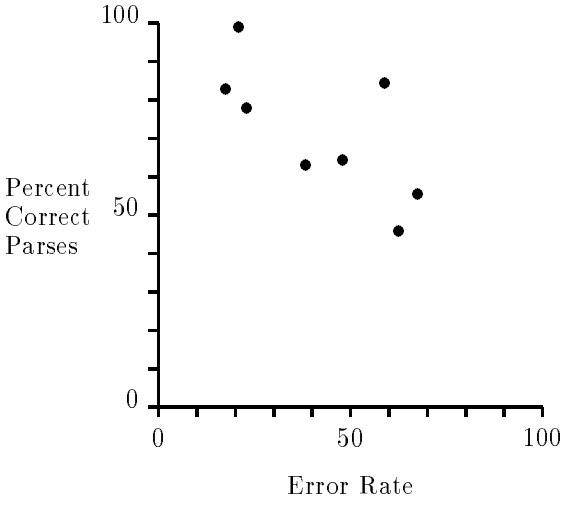

Figure 3.9: The parser accuracy on non-trivial utterances plotted against the speech recognizer error rate for each experimental subject.

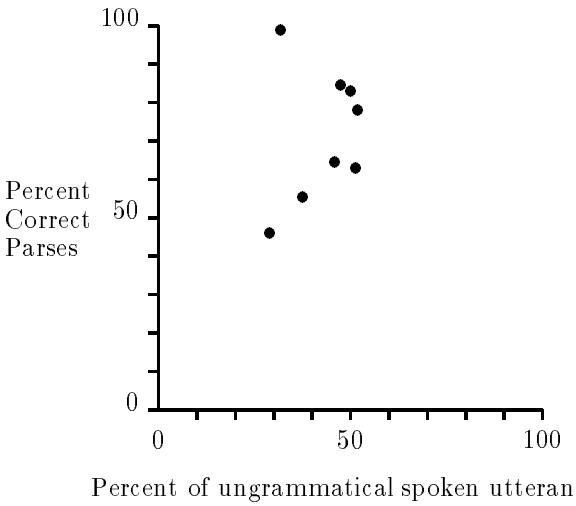

Figure 3.10: The parser accuracy on non-trivial utterances plotted against the percentage of extragrammatical spoken utterances for each experimental subject.

$$C = 0.880417M + 13.2553 \tag{3.10}$$

$$C = 97.2912 - 0.601699R \tag{3.11}$$

Figure 3.10 shows the percentage of non-trivial utterances which were correctly parsed plotted against the percentage of non-trivial spoken utterances which were not in the input language of the translation grammar. Notice that the two subjects with the lowest rate of extragrammatical utterances have respectively the worst and the best parser performance. This suggests that the accuracy of the parser and the percentage of spoken utterances which are extragrammatical are not linearly related.

3.5.5 Optimal expectation functions

The experiments were conducted using the expectation function 3.4 with $\beta = 0.98$. This choice of β was based largely on intuition and preconceived notions of what ought to work. But now, the data from the experiments allows different values of β, or even different expectation functions, to be tested experimentally.

Figure 3.11 shows the percentage of utterances which would have been parsed correctly as a function of the parameter β. When β is exactly 1.0, the expectation is effectively ignored and the final parser output is determined by selecting an MDSDT hypothesis with the minimum utterance cost. This strategy gives an overall parser accuracy of only 69%. For all other values of β greater than 0.5, however, the parser produces the much better accuracy of 83%. Furthermore, the accuracy seems not to be influenced at all by which value between 0.5 and 1.0 is chosen for β.

Figure 3.12 shows what happens when equation 3.12 is used as the expectation function and the parameter γ is varied between 0.0 and 0.97.

$$f_3(U, E) = \begin{cases} E & U \leq U_{min} + \gamma \cdot U_{max} \\ +\infty & \text{otherwise} \end{cases} \tag{3.12}$$

In this formula, U_{min} is the smallest distance associated with hypotheses output by MDSDT using the N-best approach. U_{max} is the greatest possible distance which the algorithm can produce – that is, the distance between two strings which have no words in common. The affect of expectation

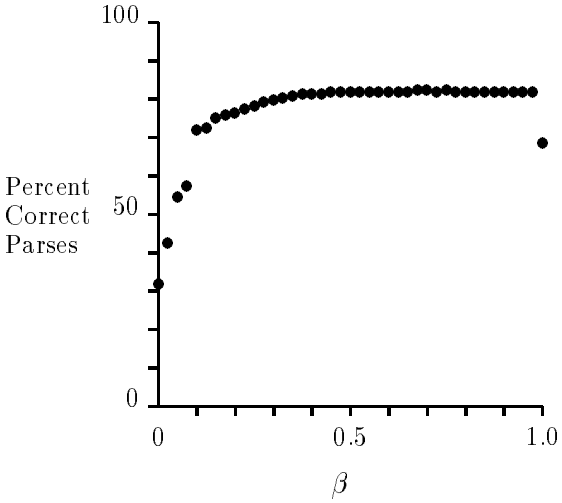

Figure 3.11: The parser's accuracy as a function of the β parameter to the expectation function of equation 3.4

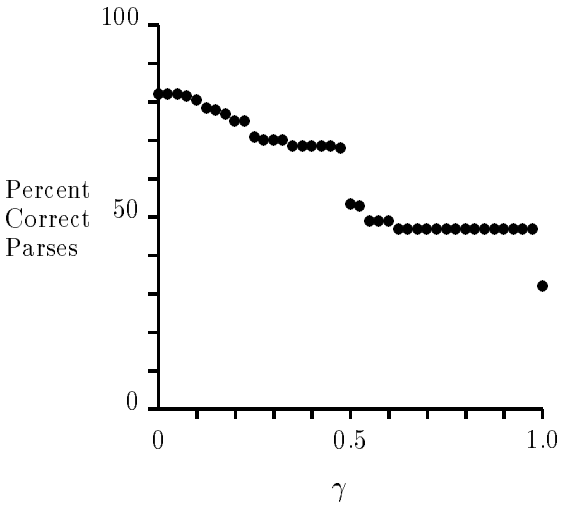

Figure 3.12: The parser's accuracy as a function of the γ parameter to the expectation function of equation 3.12

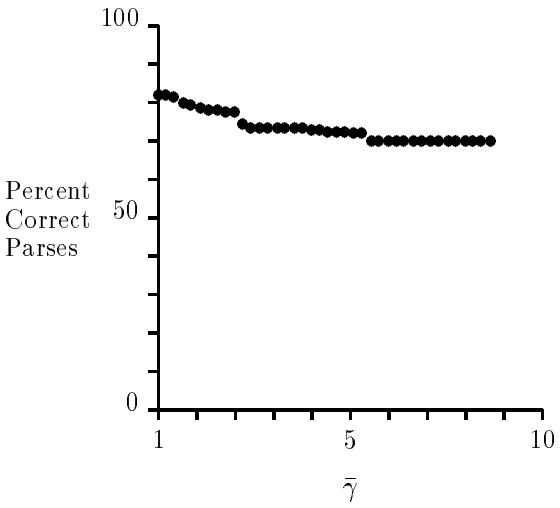

Figure 3.13: The parser's accuracy as a function of the $\bar{\gamma}$ parameter to the expectation function of equation 3.13

function 3.12 is to create a band of utterance costs around the minimum utterance cost within which all utterance costs are considered to be the same. An hypothesis is selected from hypotheses falling within this band based on their expectation cost. As can be seen in the figure, the peak parser accuracy of 83% is obtained when γ is set to exactly 0.0. Even values for γ as small as 0.005 do not perform as well as when γ is exactly 0.0. Again it appears that expectation is only helpful in deciding between two hypotheses when both have exactly the same utterance cost.

The expectation function shown in equation 3.13 was also tested.

$$f_4(U,E) = \begin{cases} E & U \leq U_{min} \cdot \bar{\gamma} \\ +\infty & \text{otherwise} \end{cases} \tag{3.13}$$

This expectation function is similar to 3.12 in that a band of allowed utterance costs are generated and expectation is used to select an hypothesis from within that band. In function 3.13, however, the band is multiplicative instead of additive. Figure 3.13 shows the parser's accuracy when equation 3.13 is used and the parameter $\bar{\gamma}$ is varied between 1.0 and 8.8. The parser accuracy is maximized at 83% correct when $\bar{\gamma}$ is set to exactly 1.0. Even a $\bar{\gamma}$ of 1.01 did not do as well. This is consistent with previous observations that expectation is only helpful when it is called upon to break an exact tie in utterance cost.

Chapter 4

Grammar Development

The need for a software-aided grammar development system (SAGDS) and an outline of what such a system might look like were discussed in section 2.2. The SAGDS which is proposed in that section implements many operations on grammars and their languages, all of which play an important role in the usefulness and functionality of the overall system. When viewed in isolation, however, many of these operations are simple tasks – tasks which are trivial to implement and which are not of theoretical interest in themselves. But three of the operations in the proposed SAGDS are interesting in their own right. The purpose of this chapter is to detail two of these three operations. (The third operations is the MDSDT algorithm discussed in the previous chapter.)

The chapter begins with a discussion of the problem of listing and counting the elements of a context-free language. It is shown that this problem is NP-hard in general, but that a solution exists which seems to work well in most practical cases. The second problem addressed by this chapter is the task of finding the subset of production rules in a context-free grammar that can be used to recognize any element of a given regular language.

The material presented in this chapter is abstract. For a description of actual implementations of the algorithms herein described, the reader is referred to sections 6.2.2 and 6.2.3 of the appendix.

4.1 Listing the Elements of a Language

Suppose one is given a CFG which is the input grammar of a SDT, and is asked the following questions:

- Does the grammar give reasonable coverage of a designated subset of the natural language?
- Does the grammar recognize any ungrammatical sentences?
- What kinds of input phrases will this grammar recognize?
- How many different input phrases (shorter than a certain number of words) will the grammar recognize?

If the CFG is the output grammar of a SDT, then similar questions might be asked:

- Does the grammar generate any logical forms that the dialog controller is unable to process?
- What are all logical forms which the grammar will generate? (The person who develops the dialog controller may want this information for testing purposes.)
- How many logical forms will the grammar generate?

These and similar questions can be answered by a program which lists or counts the strings in a CFL. This section will consider the problems of counting and listing the elements of a CFL.

The computational complexity of the listing problem is considered first. For most CFGs, the size of the output for this problem is exponential in the size of the input, so if one measures the difficulty of the problem strictly in terms of its input size then the problem will almost always require exponential time. A more reasonable approach might be to measure the difficulty in terms of the sum of the sizes of both input and output. But even with this approach, some special cases of the problem are NP-hard and thus probably require time which is exponential in the size of both input *and* output.

Since one cannot hope to (always) generate all strings in a CFL in a reasonable time, it is reasonable to ask if some of the strings can be quickly produced. Subsection 4.1.2 describes an algorithm which prints all elements of a CFL in order of length and which has proven to be efficient in practice. With this algorithm, the user could begin listing the CFL, but can interrupt the process after enough has been seen, or the user's patience has expired.

Listing the elements of a CFL is NP-hard, but this does not rule out the possibility of an efficient algorithm for counting these elements. Unfortunately, as subsection 4.1.3 will demonstrate, the counting problem is also NP-hard and is therefore no easier (asymptotically speaking) than actually listing the elements of the language.

4.1.1 Listing elements of a CFL is NP-hard

Let the problem be cast in terms of formal language recognition. Define the language CFL-LIST as follows:

$$\begin{aligned}\text{CFL-LIST} \quad = \quad & \{(G, E, u, v) \mid G \text{ is a CFG and } E \text{ is the set of all} \\ & \text{elements of } L(G) \text{ containing at least } u \text{ but no} \\ & \text{more than } v \text{ symbols}\}\end{aligned}$$

It will be shown below that the language CFL-LIST is *NP-hard*. To say that CFL-LIST is NP-hard means that finding a deterministic Turing machine which will determine whether or not its input is a string in the language CFL-LIST number of steps which is a polynomial in the number of bits needed to encode the input is at least as hard as proving that P=NP.

It is useful to first prove an intermediate result – that a related problem called *Members of Length N* (MLN for short) is NP-hard. It is not actually difficult to directly prove CFL-LIST to be NP-hard without the use of the intermediate problem MLN, but MLN finds use in another problem later in this chapter, so it seems expedient to define and prove it here. MLN is defined thus:

$$\begin{aligned}\text{MLN} \quad = \quad & \{(G, N) \mid G \text{ is a CFG and } L(G) \text{ contains at} \\ & \text{at least one string of length } N\}\end{aligned}$$

Theorem 4.1 *MLN is NP-hard.*

Proof: The problem SUBSET-SUM has been proven to be NP-complete in [21]. It will be shown that SUBSET-SUM can be reduced to MLN in polynomial time. SUBSET-SUM asks if there is a subset of a given set of integers which sums to a given integer. Formally:

$$\begin{aligned}\text{SUBSET-SUM} \quad = \quad & \{(A, N) \mid A = \{a_1, a_2, \ldots, a_p\} \text{ and there exists} \\ & B \subseteq A \text{ such that } \sum B = N\}\end{aligned}$$

The reduction begins by constructing a grammar G in the MLN problem based on the set A of SUBSET-SUM. The grammar will be constructed so that the language generated will be strings of all "1"s and the length of each string in the language will be the sum of the elements in some subset of A. The start-state of the grammar will be S and will generate a string of nonterminals, one nonterminal for each element in A.

$$S \rightarrow A_1 A_2 A_3 \ldots A_p$$

Also define a set of k nonterminals, each of which produce a string of 2^k "1"s. The integer k will be the floor of the base 2 logarithm of the largest element in the set A.

$$\begin{aligned} D_1 &\rightarrow 1 \\ D_2 &\rightarrow 11 \\ D_3 &\rightarrow D_2 D_2 \\ D_4 &\rightarrow D_3 D_3 \\ &\vdots \\ D_k &\rightarrow D_{k-1} D_{k-1} \end{aligned}$$

Finally, each of the nonterminals A_1 through A_p will produce either the empty string, λ, or a sequence of 1s equal in length to the value of the corresponding element of the set A. This sequence is not generated directly but is produced by an appropriate combination of the nonterminals D_1 through D_k. As an example, suppose $a_1 = 23$. This results in :

$$A_1 \rightarrow \lambda \mid D_5 D_3 D_2 D_1$$

Using this construction, a given A may be easily converted to a corresponding G in linear time. It is also apparent that G will generate all and only strings of length equal to the sum of elements in subsets of A and that the size of the encoding of G is linearly related to the encoded size of A. Hence, to determine if any pair (A, N) is an element of SUBSET-SUM, one need only convert A into G and ask if the pair (G, N) is an element of MLN, for the language $L(G)$ will contain an element of length N if and only if there is a subset of A which sums to N. This completes the reduction of SUBSET-SUM to MLN, and thus the proof that MLN is NP-hard. ■

The fact that MLN is NP-hard will now be used in a proof that CFL-LIST is NP-hard.

Theorem 4.2 *CFL-LIST is NP-hard.*

Proof: The opposite problem of MLN is co-MLN. More specifically

$$\begin{aligned} \text{co-MLN} = \{&(G, N) \mid L(G) \text{ is a CFG containing no} \\ &\text{elements of length } N\} \end{aligned}$$

Because MLN is NP-hard so is co-MLN. But, an instance (G, N) of co-MLN is exactly the same as the instance $(G, \emptyset, N, N)$ of CFL-LIST where $\emptyset$ means the empty set. Hence CFL-LIST must also be NP-hard. ■

4.1.2 An Algorithm for Listing Elements of a CFL

The above proof that CFL-LIST is NP-hard is weak on two accounts:

1. The proof relies on the NP-completeness of SUBSET-SUM, but SUBSET-SUM is only weakly NP-complete. Unless some of the element sizes of SUBSET-SUM are exponentially large, there exists an efficient dynamic programming solution to the problem. Translating this weakness to CFL-LIST means that the value of v must become large before the problem becomes unmanageable.

2. CFL-LIST has only been proven to be NP-hard for the special case where $u = v$. In practice, this is almost never the case. One usually sets u equal to 1 and v equal to some value in the range of 10 to 20.

For these reasons, it seems likely that one should be able to find an algorithm for solving CFL-LIST which, though infeasible to compute for certain special cases, is adequately efficient for many practical instances of the problem.

The following algorithm is able to compute all strings in a context-free language, and experience shows that it usually does so with acceptable efficiency. Furthermore, the shortest strings are computed first. With this strategy, the user is able to begin listing the strings of the language, but later to stop the listing if it appears that computing the complete list will take too long.

Algorithm 4.1 *A procedure for computing the elements of a CFL beginning with the shortest.*

Input:

1. G, a context-free grammar.
2. u and v, the length of the shortest and longest elements of $L(G)$ which are to be printed.

Output:

1. All strings in $L(G)$ which have at least u symbols but no more than v symbols. The shortest strings are printed first.

Method:

1. Let the nonterminal symbols in G be $\aleph_1, \aleph_2, \ldots, \aleph_p$. The start symbol of G is $\aleph_1$. Let $L_{i,j}$ be a set of terminal strings each containing exactly j symbols and which are generated from nonterminal symbol $\aleph_i$. Mark $L_{i,j}$ as UNCOMPUTED for all $1 \leq i \leq p$ and $1 \leq j \leq v$.
2. For every i between u and v do the following:
   (a) If $L_{1,i}$ is marked UNCOMPUTED then use algorithm 4.2 to compute it.
   (b) Print every element of $L_{1,i}$. (All elements of $L_{1,i}$ are, of course, the same length, so the requirements of the algorithm are satisfied no matter what order they are printed. Yet, the algorithm will give a more pleasing output if the elements are printed in, say, alphabetical order.)

Algorithm 4.2 *A procedure which computes all terminal strings of a given length which can be derived from a given nonterminal.*

Input:

1. G, a context-free grammar.
2. $\aleph_i$, a nonterminal in G.

3. n, the length of every string in the output

Output:

1. $L_{i,n}$, a list of every phrase which can be derived from $\aleph_i$ in G and which contains exactly n symbols.

Method:

1. Mark $L_{i,n}$ as COMPUTED. This will block an infinite recursion which occurs if $\aleph_i \Rightarrow^* \aleph_i$ is a valid derivation in G.
2. Allocate a phrase template g which has slots to hold n symbols.
3. For every production rule r in G which has $\aleph_i$ as its LHS, call algorithm 4.3 with $L_{i,n} = L_{i,n}$, $r = r$, $x = 0$, $w = 0$, and $g = g$.
4. Return $L_{i,n}$.

Algorithm 4.3 *A procedure which computes a single terminal string of a given length which can be derived from a given nonterminal.*

Input:

1. $L_{i,n}$, a list of phrases which is currently being computed.
2. r, a production rule in G.
3. x, the number of RHS symbols in r which have already been matched.
4. w, the number of symbols which have already been inserted into g.
5. g, a template for a new phrase.

Output:

1. Additional phrases are (usually) added to $L_{i,n}$ by this procedure.

Method:

1. If x is less than the number of symbols on the RHS of r and the $(x+1)$-th RHS symbol of r is a terminal and w is less than n do this:
   (a) Put the $(x+1)$-th RHS symbol of r into the w-th symbol slot in g.
   (b) Increment x.
   (c) Increment w.
2. If x equals the number of symbols on the RHS of r and w equals n then the template g is complete. Add g to $L_{i,n}$ and return.
3. If x is less than the number of symbols on the RHS of r and the $(x+1)$-th symbol is the nonterminal $\aleph_j$ then do the following:
   (a) Let r and s be respectively the number of terminal and nonterminal symbols on the RHS of r which are to the right of the $(x+1)$-th symbol.
   (b) Let the upper bound $\top \leftarrow n - w - r$. $\top$ is the maximum number of symbols which $\aleph_j$ can be expanded into if the complete phrase is to be no greater than n symbols long.

(c) $-$ is the minimum number of symbols which $\aleph_j$ can be expanded into if the complete phrase is to be no less than n symbols long. If s is zero set $- \leftarrow \top$. Otherwise, set $- \leftarrow 0$.

(d) For every $- \leq k \leq \top$ do the following:

i. If $L_{j,k}$ is marked UNCOMPUTED, then use algorithm 4.2 to compute it.

ii. For each string α in $L_{j,k}$ do this:

A. Append α to g beginning at the w-th slot.

B. Recursively call this algorithm with $L_{i,n} = L_{i,n}$, $r = r$, $x = x$, $w = w + k$, and $g = g$.

4.1.3 Counting the Elements of a CFL

Counting the number of elements in a CFL is no easier than listing elements in the following sense: both problems are NP-hard. To prove that the counting problem is NP-hard, it will first be stated in terms of language recognition.

$$\begin{aligned} \text{CFL-SIZE} \quad = \quad & \{(G, L, N) \mid G \text{ is a CFG and there are exactly } N \\ & \text{elements in } L(G) \text{ which are no more than } L \\ & \text{symbols long}\} \end{aligned}$$

Theorem 4.3 *CFL-SIZE is NP-hard.*

Proof: Using the technique of binary search, an implementation of CFL-SIZE can be invoked a polynomial number of times to compute N given G and L. It is possible to determine whether or not $L(G)$ contains any string of length exactly L by first computing N_1 from G and L and then computing N_2 from G and $L - 1$ and finally comparing N_1 to N_2. But this is a method for computing MLN. In other words, MLN can be reduced to a polynomial number of invocations of CFL-SIZE. But MLN is NP-hard (by theorem 4.1) so CFL-SIZE must also be NP-hard. ■

4.2 Extracting Relevant Subsets of a Grammar

In the course of studying or revising a large syntax-directed translation grammar, or even an large context-free grammar, it is useful to extract those production rules which relate to the feature or behavior of interest and group these rules together for easy reference. But in a large grammar and without some kind of automated tool for extracting the relevant parts of the grammar, one is likely to spend an inordinate amount of time searching for the relevant production rules. Furthermore, one might never be confident that all of the pertinent production rules have been found.

This section will describe an algorithm for extracting relevant production rules from a CFG. Call this algorithm the *relevant subset algorithm*. A language of interest is specified by a regular expression. Those production rules in the CFG are extracted which can participate in any successful parse of a string in the language of interest.

Algorithm Overview

The operation of the algorithm is intuitively simple. From the regular expression, a finite automaton called the *input graph* is constructed which will accept the language of the regular expression and which has the following properties:

1. The automaton has no null transitions.

2. The automaton has only a single accepting state.

Such an automaton can always be constructed provided that the language of the regular expression does not contain the empty string. The automaton may not be deterministic, but that is of no consequence to the overall algorithm.

After the automaton is constructed, it becomes the input to a special parser, called the *Pertinent Production-rule Parser* (PPP), that determines which production rules to include in the extracted subset.

4.2.1 Prior Art

Algorithms for converting regular expressions into graphs are described in [19]. The operation of PPP is similar to Earley's general parsing algorithm [8], but with some simple enhancements for tracking production rule subsets. Thus neither of the major components of the following algorithm are novel. However, the way in which the components are combined and used does appear to be unique.

4.2.2 Construction of the Input Graph

To convert the regular expression into an appropriate finite state automaton, use algorithm 4.4.

Algorithm 4.4 *Conversion of a regular expression into a nondeterministic finite automaton without null transition and with only one accepting state.*

Input:

1. A regular expression which does not accept the empty string.

Output:

1. A nondeterministic finite state automaton with no null transitions and with only a single accepting state and which accepts the same language as the input regular expression.

Method:

1. The regular expression is expressed as a string of symbols which can be defined by a CFL. Use a context-free parser to discover the structure of the regular expression and to express that structure as a tree.
2. Create the starting state and the accepting state of the finite automaton. Designate these states numbers 0 and 1.
3. Call algorithm 4.5 with the regular expression tree and with the two states 0 and 1 in order to convert the regular expression into a finite automaton with null transitions.
4. Call algorithm 4.6 to convert the finite automaton with null transitions into one without null transitions.

The following algorithm recursively walks the regular expression tree in order to construct a finite automaton which accepts the same language. The resulting finite automaton is correct (as can be proven by a simple induction) but it contains many null transitions.

Algorithm 4.5 *Conversion of a regular expression into a nondeterministic finite automaton with null transitions and only one accepting state.*

Input:

1. A regular expression represented as a tree.
2. A number, a, identifying a node in the finite automaton which should become the start state.
3. A number, z, identifying a node in the finite automaton which should become the accepting state.

Output:

1. A nondeterministic finite automaton with null transitions and with start and accepting states identified in the input and which accepts the same language as the input regular expression.

Method:

1. If the regular expression is composed of the concatenation of two other regular expressions, α and β, then do the following:
   (a) Create a new node in the finite automaton identified by the number x.
   (b) Call this algorithm recursively to build an automaton for α between nodes a and x.
   (c) Call this algorithm recursively to build an automaton for β between nodes x and z.
2. If the regular expression is composed of the alternation of two other regular expressions, α and β, then do the following:
   (a) Call this algorithm recursively to build an automaton for α between nodes a and z.
   (b) Call this algorithm recursively to build an automaton for β between nodes a and z.
3. If the regular expression is the Kleene closure of another regular expression α, then do the following:
   (a) Create a two new nodes in the finite automaton identified by the numbers x and y.
   (b) Call this algorithm recursively to build an automaton for α between nodes x and y.
   (c) Insert a null transition from y to x.
   (d) Insert a null transition from x to y.
   (e) Insert a null transition from a to x.
   (f) Insert a null transition from y to z.
4. Finally, if the regular expression is the symbol symbol c then simply construct an arc labeled with c between nodes a and z.

The next algorithm is used to remove the null transitions from a finite automaton.

Algorithm 4.6 *A procedure for removing all null transitions from a nondeterministic finite automaton.*

Input:

- A nondeterministic finite automaton with null transitions and with only a single accepting state.

Output:

- A nondeterministic finite automaton with only a single accepting state but without null transitions.

Method:

- For every ordered pair (x, y) where x and y are nodes in the automaton connected by a null transition, do the following:
  1. For every arc in the automaton from u to v and labeled by symbol p do the following:
     (a) If $v = x$ then create a new arc from u to y and labeled with p if no such arc already exists.
- Keep repeating the previous step until no new arcs can be added.
- Remove all null transitions from the automaton.

4.2.3 The Pertinent Production-rule Parser

The algorithm for PPP is reminiscent of the MDP parsing algorithm described in section 3.1. R-arcs and NT-arcs are added to the input graph until an NT-arc is found which spans the start state and the accepting state of the input graph and which is labeled by the start symbol of the grammar. The differences between the algorithms are these:

- The R-arcs and NT-arcs of PPP are labeled with subsets of the grammar, not with distances and semantics as in MDP or MDSDT.
- The input graph for PPP can contain cycles.
- PPP is not concerned with ungrammatical or ill-formed input.

The final output of the relevant subset algorithm is found by calling the PPP algorithm to compute the NT-arc which is labeled by the start nonterminal for the grammar and which spans the start state and the accepting state of the input graph. The subset of production rules associated with the NT-arc is the relevant subset algorithm's result.

Algorithm 4.7 *The Pertinent Production Rule Parser*

Input:

1. $\aleph$, a nonterminal of the grammar.
2. a and z, two nodes from the input graph.

Output:

1. The set of production rules from the grammar which can be used to derive any sequence of words connecting nodes a and z from the nonterminal $\aleph$.

Method:

1. If the NT-arc $(\aleph, a, z)$ has already been computed then return the production rule set attached to that arc without further computation.
2. Create a new NT-arc $(\aleph, a, z)$ and initialize its production rule set to the empty set.

3. For every production rule r which as $\aleph$ has its LHS, do the following:
   (a) Use algorithm 4.8 to find the production rule set associated with the R-arc $(r, |r|, a, z)$.
   (b) Add every production rule in the R-arc rule set of the previous step to the rule set for the NT-arc $(\aleph, a, z)$.
4. Return the production rule set for the NT-arc $(\aleph, a, z)$.

Algorithm 4.8 *Computation of R-arcs for the pertinent production rule parser.*

Input:

1. r, a production rule the grammar.
2. x, an integer between zero and $|r|$ the number of symbols on the RHS of the production rule.
3. a and z, two nodes from the input graph.

Output:

1. The set of production rules from the grammar which can be used to derive any sequence of words connecting nodes a and z from the first x symbols on the RHS of production rule r.

Method:

1. If the R-arc (r, x, a, z) has already been computed then return the production rule set attached to that arc without further computation.
2. Create a new R-arc (r, x, a, z) and initialize its production rule set to the empty set.
3. If $x = 0$ then add r to the production rule set on the R-arc and skip ahead to step 7.
4. Call the x-th symbol on the RHS of r $\aleph$.
5. If $\aleph$ is a nonterminal then do the following
   (a) For every node v in the input graph, use algorithm 4.7 to find the set of production rules for the NT-arc $(\aleph, v, z)$.
   (b) If the production rule set for $(\aleph, v, z)$ is not empty the compute also the production rule set for the R-arc $(r, x-1, a, v)$.
   (c) If the production rule set for $(r, x-1, a, v)$ is also not empty, then add all production rules associated with the NT-arc $(\aleph, v, z)$ and the R-arc $(r, x-1, a, v)$ to the production rule set for the R-arc (r, x, a, z).
   (d) Go back to step 5a until all possible values v have been tested.
6. If $\aleph$ is a terminal then do the following
   (a) Let v be a node of the input graph such that there is a T-arc from v to z labeled by the terminal symbol $\aleph$.
   (b) Compute the production rule set for the R-arc $(r, x-1, a, v)$.
   (c) If the production rule set for $(r, x-1, a, v)$ is not empty then add all production rules in that set to the production rule set for the R-arc (r, x, a, z).
   (d) Go back to step 6a until all possible values v have been tested.
7. Return the production rule set for the R-arc (r, x, a, z).

Chapter 5

Verification

This chapter describes details of the verification subsystem which was introduced and overviewed in section 2.3. The bulk of this chapter is taken up by section 5.1 which describes techniques for deciding whether or not to initiate a verification subdialog. In the next section, the benefits of verifying semantics instead of just verifying syntax are enumerated. A description of an actual implementation of the verification subsystem is undertaken in section 5.3, followed by an analysis of experimental results and two annotated examples of verification subdialogs in sections 5.4 and 5.5.

5.1 Deciding When To Verify

The decision to verify a particular input is governed by two factors:

1. the likelihood that the meaning deduced by the parser is what the user intended to communicate, and

2. the importance of the meaning to the success of the overall dialog.

A simple method of incorporating both of these factors into the verification decision is as follows: Assign to each meaning a *confidence* which is an estimate of the chance that the meaning was correctly deduced from the user's original speech, and a *verification threshold* which is a measure of the importance of the deduced meaning to the success of the dialog. Initiate a verification if and only if the confidence is less than the verification threshold. Methods for estimating the confidence and verification threshold are described in the following two subsections.

5.1.1 Confidence Estimates

Several algorithms for estimating the confidence of a particular meaning have been developed and compared in an effort to find one general method which provides consistently good results. All of the algorithms tested compute the confidence as a linear function of four measurements:

Total Error The *total error* of a particular utterance is the amount of deviation between the user's speech and the most similar speech template. The deviation can be measured at both the speech recognition level (how closely does the user's speech match internal speech templates) and at the grammatical level (how closely does the sequence of words output by the speech recognizer match a well-formed phrase in the grammatical model of the input language.)

In the implementation, the total error is taken to be the total cost computed by the expectation function. (See section 3.4.5.) Deviation at the speech recognition level could not be used because requisite information is not available from the commercial speech recognizer used in the implementation.

Normalized Error The *normalized error* is the amount of deviation per fixed unit of input. The normalized error is therefore just the total error divided by the size of the user's input. The size of an input might be its temporal length, or the number of words in the phrase, or some other appropriate measure. In this implementation, the size of an utterance is the number of words output by the speech recognizer.

Expectation Cost The *dialog expectation* is a set of meanings which the dialog controller anticipates the user will try to communicate. Each meaning has an associated *expectation cost* which is a measure of how strongly that particular meaning was anticipated. (See section 3.4.1.)

The algorithm used to compute the dialog expectation in the implementation is described fully in [35]. Here is an intuitive overview of that algorithm: The state of a dialog is viewed as a sequence of nested contexts, each more specific and focused than its predecessors. Meanings which relate to the innermost context are given the lowest expectation. Meanings which can only be interpreted in the first enclosing context are given a slightly higher expectation cost. This progression continues, so that meanings which can only be understood in more remote contexts are given comparatively higher expectations cost.

Distinctness The *distinctness* is a measure of ambiguity in an utterance. A high distinctness shows that the meaning deduced by the speech understander is the only reasonable interpretation of the input. A low distinctness indicates that there are several competing interpretations of the user's speech, and that the meaning output by the speech understander is only one of these interpretations.

In this paper, distinctness is computed as the difference between the total error of the meaning and the total error of that meaning which was the parser's second choice.

Over- and Under-verifications

Here are two important definitions: An *under-verification* is defined as the event where the parser generates a meaning which is incorrect but which is not verified. An *over-verification*, on the other hand, occurs when a correct meaning is verified. An under-verification results in a misunderstanding, but an over-verification is only a vexation to the user. The goal of any confidence estimating function is to simultaneously minimize the number of both under- and over-verifications. It is usually the case, however, that under- and over-verifications trade off against one another, so that one may decrease the number of under-verifications only by increasing the number of over-verifications, and vice versa. Hence, it is instructive to think of over-verifications as the price one pays for reducing the number of under-verifications.

The performance or "goodness" of a particular confidence estimating function can be visualized by plotting the number of under-verifications versus the number of over-verifications which occur using the given confidence estimator, for various thresholds, over a fixed set of meanings. Figure 5.1 shows the performance curve for a particular confidence estimating function named α. The function α computes the confidence as a linear combination of normalized error and dialog expectation, weighted so that a single level change in context is approximately equal to inserting or deleting a single content word in a twenty-word long utterance. Each point in figure 5.1 represents the number of over- and under-verifications which would result if a single threshold value were used

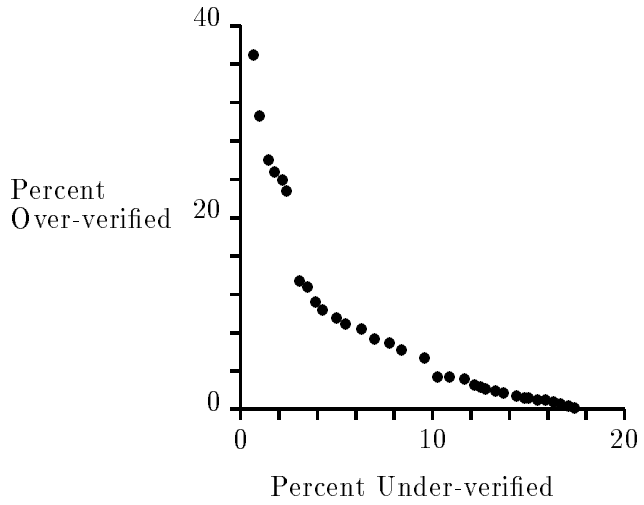

Figure 5.1: Performance curve for the confidence estimating function named α

in conjunction with confidence estimator α in order to decide whether or not to verify each of a standard set of meanings. The over- and under-verifications are expressed as a percentage of the total number of meanings analyzed. The data used to generate figure 5.1 is the 2804 utterances and their meanings recorded during the experiments described in section 3.5.

Comparison of various confidence estimating functions

Two confidence estimators may be compared by plotting their performance curves on the same graph. A confidence estimator is normally judged to be better if it consistently has a smaller over-verification rate, for any given under-verification rate. Figures 5.2 through 5.5 compare α to confidence estimators based on the four fundamental confidence measurements mentioned above. The α function is used in these comparisons since it has been found to provide the best results under most circumstances. Other confidence estimating functions which were compared to α and which were found to be inferior are these:

- Functions built from a linear combination of normalized error and total error.
- Functions built from a linear combination of total error and the sum of normalized error and dialog expectation.
- Functions which select the greater of normalized error and weighted total error.
- Functions which select the lesser of normalized error and weighted total error.

All the performance curve plots displayed in this these are derived from the 2804 utterances collected from the experiments described in section 3.5. However, this was not the only dataset used in the actual analysis. Other datasets include:

- The 2804 utterances in the experiments, but with duplicate occurrences of the same input removed. This reduced the number of utterances to 1134.
- The 2804 utterances in the experiments, but processed using the second implementation of the parser, and using a heavily revised version of the grammar.

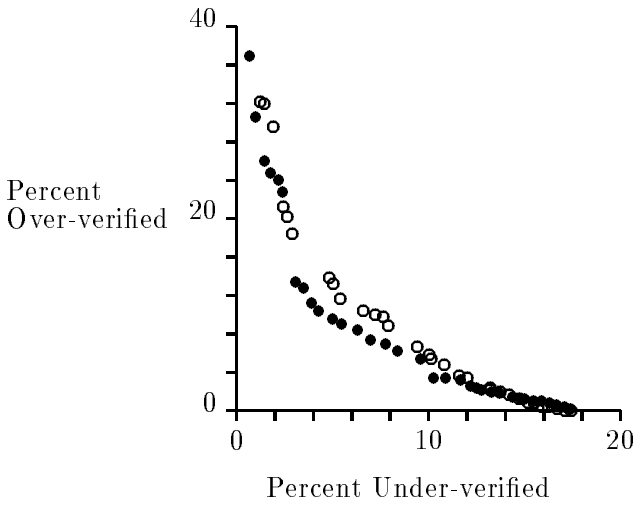

Figure 5.2: Performance curve for the confidence estimator based on total error alone ($\circ$) versus the confidence estimator α ($\bullet$).

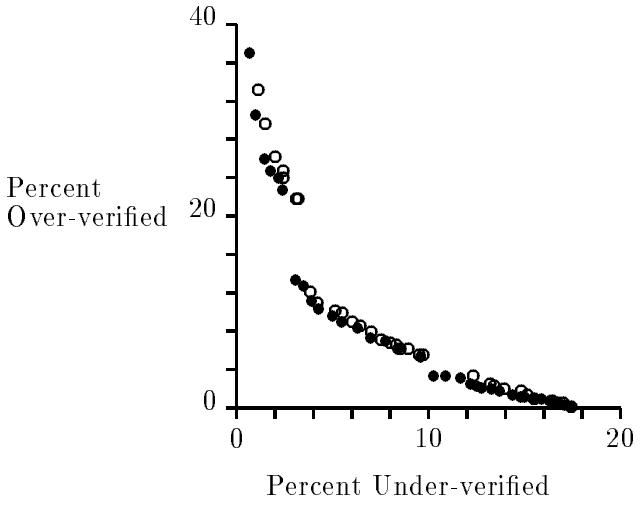

Figure 5.3: Performance curve for the confidence estimator based on normalized error ($\circ$) versus the confidence estimator α ($\bullet$).

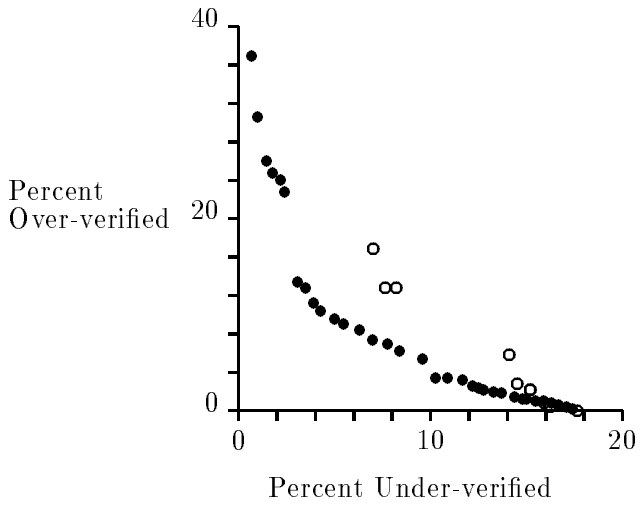

Figure 5.4: Performance curve for the confidence estimator based on only dialog expectation (∘) versus the confidence estimator α (•).

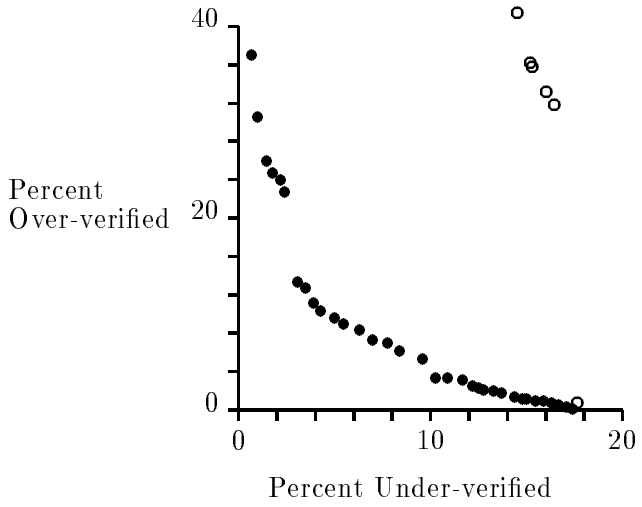

Figure 5.5: Performance curve for the confidence estimator based on only distinctness (∘) versus the confidence estimator α (•).

- The data specified in the previous bullet but with duplicate entries removed.
- Subsets of the data specified in all of the previous bullets consisting of only utterances spoken by one of the eight experimental subjects.

In every case, the performance plots were scaled differently to reflect the differing speech recognition and parsing accuracy of the various data sets, but the basic shape of the performance curves was preserved. Though in some cases other confidence estimating functions did just as well, none were better than the function α.

5.1.2 Selecting A Verification Threshold

The second concern in deciding whether or not to verify a given meaning is the selection of a verification threshold against which to compare the confidence estimate. The verification threshold is the quantity which is used to tune the propensity of the system to verify. A low verification threshold means that verifications will be frequent, to the point of being bothersome, but that the effective misunderstanding rate will be low. A high verification threshold will result in infrequent verifications, but a higher misunderstanding rate.

It is important to note that the verification threshold need not be a constant; it may depend on the meaning deduced by the speech understander. Meanings which are critical to the success of the dialog may be given a high threshold, thus requiring that there be high confidence in the accuracy of the meaning to avoid verification, whereas superficial or unimportant meanings may be given a low threshold, so that the system will not waste the user's time verifying unimportant details.

The selection of an appropriate verification threshold is highly dependent upon the requirements of the dialog system, and as a result, it is difficult to make general statements about which verification thresholds are appropriate and which are not. In many cases, however, the selection of a verification threshold might be aided by a *dialog work* analysis. Dialog work is an intuitive concept which can be thought of as roughly the number of user-machine interactions or the amount of time needed to complete a dialog. The analysis which uses dialog work to select a verification threshold is as follows:

A particular meaning, m, is computed by the speech understander, and a verification threshold is wanted for this meaning. Let V be the amount of dialog work needed to verify m, and let E be the amount of extra dialog work needed to recover if m is a misunderstanding and is not verified. Let u be the probability of under-verification, and let $f(u)$ be the corresponding over-verification probability. Assume that u and $f(u)$ can be computed from historical under- and over-verification rates, so that $f(u)$ is actually the performance curve for the chosen confidence estimator. Let the dialog work required after m is processed be K. If we assume that a verification subdialog will always find the correct meaning, then the total dialog work required for a given under-verification rate u is $W(u)$ in the following equation:

$$W(u) = uE + f(u)V + K \tag{5.1}$$

If we could compute the value of u in equation 5.1 which minimized the dialog work, then this u could be used in conjunction with the confidence estimator's performance curve to find the corresponding verification threshold. The desired u can be found by solving the equation

$$\frac{d}{du}W(u) = 0 \tag{5.2}$$

The solution is

$$\frac{d}{du}f(u) = -\frac{E}{V} \tag{5.3}$$

Equation 5.3 says that the under-verification rate which minimizes the dialog work is the ordinate of the point on the performance plot of the confidence estimator function which has a slope of $-E/V$. Here is the key observation: the performance plots of many confidence estimators contain a small region which has a large second derivative. Call such a region a *knee* of the graph. For example, there is a knee in α in the vicinity of $u = 0.03$. For many values of $-E/V$, the point on the performance graph which has a slope of $-E/V$ will be on the knee of the graph. Hence, the under-verification rate which minimizes the dialog work will likely be in the knee of the performance graph.

It is not difficult to find objections to the above analysis. Perhaps the most serious are that V is not constant but rather depends on u, and that a verification subdialog does not guarantee that the intended meaning will eventually be found. Nevertheless, the above analysis is a reasonable first-order approximation.

5.2 The Benefits of Verifying Meanings Instead of Syntax

There are several benefits to be derived from verifying at the semantic level as opposed to the syntactic level. Below is listed a few of these advantages:

- Operation at the semantic level allows the verification prompt (the statement used by the computer to begin a verification subdialog) to be a paraphrase of what the user originally said. The verification prompt can expand pronouns into their antecedents, fill out elliptic phrases and words, and make use of synonyms.
- Since meaning, not syntax, is what is verified, the system is able to respond robustly to errors which arise from sources other than the speech recognizer.
- Because the meaning of the utterance is available, the verification threshold used can be dynamically adjusted to better reflect the importance of the meaning to the success of the dialog.
- Dialog expectation, which is usually only available in semantic form, can be used in the confidence estimate if verification occurs on semantics instead of syntax.

5.3 An Implementation

The techniques described in section 5.1 have been used to implement a meaning verification subsystem. This subsystem was installed in the SLS used for the experiments described in section 3.5. In reference to figure 1.1, the new verification subsystem was inserted between the parser and the dialog controller. The verification subsystem intercepts the meaning which the parser would normally send to the dialog controller. If no verification of the intercepted meaning is indicated, then the meaning is relayed to the dialog controller unaltered. If, however, the meaning requires verification then the verification subsystem will speak an appropriate prompt by communicating directly with the speech synthesizer, interpret the users response as understood by the parser, and finally send a corrected meaning to the dialog controller. The dialog controller is never aware that a verification has occurred.

A sketch of an algorithm used in the verification subsystem is as follows:

1. Listen for an input from the user. Parse the input and call the deduced meaning X.

2. If X does not need to be verified then send X on to the dialog controller and exit.
3. Compute a natural language paraphrase of X and call it Y.
4. If Y could not be computed for any reason, then say "I'm not sure what you mean. Please repeat." and go back to step 1.
5. If Y was successfully computed then say "Did you mean to say Y".
6. Add meanings for "yes" and "no" to the dialog expectation. Listen for the user to respond and parse the new response into Z.
7. If Z also needs to be verified, then say "I still can't understand what you said. Please repeat your original statement." and go back to step 1.
8. If Z means "yes" then send X to the dialog controller and exit.
9. If Z means "no" then say "Please repeat what you meant to say" and go back to step 1.
10. If Z means neither "yes" nor "no" then transmit the meaning Z to the dialog controller and exit.

The decision to verify is made by comparing the confidence estimate and verification threshold for each utterance. The verification subsystem estimates the confidence using function α. The verification threshold is dynamically selected based on the meaning. Questions and some statements which do not affect the future course of the dialog are given a very low verification threshold so that they are effectively never verified. This is done because the cost of verifying such an input is never less then the cost of recovering if such an input happens to be a misunderstanding. For statements which do influence the future course of the dialog, a verification threshold in the knee of the performance curve is selected.

5.4 Experimental Results

The verification subsystem was not implemented in time to be used in the experiments described in section 3.5. Nevertheless, the data gathered in these experiments can be used to estimate how well the verification subsystem would have improved the parser's performance had the subsystem been operational.

Assuming a constant verification threshold which is adjusted so that the smallest possible under-verification rate is without an "unreasonable" over-verification rate and assuming that every verification subdialog will eventually result in the discovery of the correct meaning, then the percentage of all utterances which are correctly parsed is raised from 83% (shown in table 3.8) to 97% as shown in table 5.1. The corresponding over-verification rate is shown in table 5.2. Both of these tables are given in the same format as the reported performance results of the speech recognizer and parser in sections 3.5.3 and 3.5.4. The values in the smaller typeface are for data collected from a single experimental subject and the single value in the larger typeface is the aggregate value.

Tables 5.3 and 5.4 are the counterparts to tables 5.1 and 5.2, respectively, where only non-trivial utterances are considered. (Recall from section 3.5.4 that a non-trivial utterance is one which consists of more than one word.)

99.4	99.4	98.5	97.3	**96.9**	96.5	96.2	95.8	92.0

Table 5.1: Percentage of all inputs which would be correctly parsed after verification

63	36	23	**22**	15	15	11	9	7

Table 5.2: The over-verification rate corresponding to table 5.1

5.5 Examples

Below are two example verification subdialogs, together with commentary. Each of the verifications occurred during a dialog which is part of a videotaped demonstration [18].

Prior to the first verification subdialog, the system has asked the user to describe the state of a certain seven-segment LED, which the user did. An adjustment was made to the circuit and the system again asks for a description of the LED.

computer: What is the LED displaying?
user: The same thing.
computer: Did you mean to say that the LED is displaying the same thing?
user: Yes.

The user's response "the same thing" was misrecognized as "the same same". The parser used the words "the same" to guess that the object of the user's intended meaning was "the same thing" and used expectation from the dialog controller to fill in the missing subject and verb "the LED is displaying". However, there was significant deviation between the phrase input to the parser and any well-formed phrase of the same meaning. In particular, 1 in 3 input words were wrong. This reduced the estimated confidence below the verification threshold and triggered the verification subdialog. Notice that the subject and verb of the meaning, omitted from the user's original utterance, are included in the verification prompt. This is only possible because the verification occurs on semantics instead of syntax. The user responds to the verification prompt in the affirmative, and so the original meaning is transmitted to the dialog controller. This verification subdialog would count as an over-verification.

Here is another verification subdialog with a similar context. The resolution of this subdialog is very different, however.

computer: What is the LED displaying?
user: The one is flashing for a longer period of time.
computer: Did you mean to say the LED is flashing a one?
user: The LED is flashing one and seven.

99.0	98.9	98.6	96.8	**95.0**	94.2	93.0	92.1	88.2

Table 5.3: Percentage of non-trivial utterances which would be correctly parsed after verification

44	40	29	28	**25**	18	16	16	14

Table 5.4: The over-verification rate corresponding to table 5.3

In this dialog segment, the user's first input contains the word "period" which is not in the speech recognizer' vocabulary and is therefore omitted from the speech recognizer's output. Other words in the input phrase are also misrecognized. The output of the speech recognizer is: "one is flashing forth longer in a time." The parser is unable to make any sense out of the phrase "forth longer in a time", but it does know what "one is flashing" means. The parser combines these three words with dialog expectation from the dialog controller to conclude that the user meant to say that the LED is flashing the digit one. However, five of the eight words in the user's utterance deviate from any valid syntax for this statement. This resulted in a large normalized error which triggered the verification subdialog. The user's second statement is different from the first. This might have happened because the user realized that "period" was not in the vocabulary and that she would need to rephrase her remark. Whatever the reason for the change, the second statement by the user was parsed without significant error and its meaning was transmitted to the dialog controller in place of the first meaning.

Chapter 6

Implementation

This chapter describes actual implementations of the subsystems previously mentioned in this thesis – the parser, the grammar development system, and the verification subsystem. Three goals of this chapter are as follows:

1. It is hoped that the programs written in support of this thesis will be useful for future research both by the author and by others. To this end, the programs have been designed and written in a very portable manner so that they may be connected to diverse and divergent SLSs with a minimum of rework. The descriptions provided here are intended to provide a *Users Reference Manual* for the implementations.

2. Some of the implementation techniques used are novel and have worked very well in that they have provided a stable and robust structure to the code which is also easily altered and modified. This is especially the case with the parser which is implemented as 6 separate processes under UNIX which communicate through pipes. Future researchers may wish to imitate these techniques even if they don't use the actual code.

3. Sometimes, it is easier to understand theoretical discussions if the reader has a clear vision of the end result. Hence, some may wish to read this before perusing the more theoretical issues in the earlier chapters.

6.1 The Parser

This section documents the second implementation of the parser. The parser runs as six separate processes under UNIX.

`mdt` This is the kernel of the parser. It executes the MDSDT and dialog expectation matching algorithms. This process is described further in subsection 6.1.1.

`FromProlog` This process receives dialog expectation information from the dialog controller (which is coded in Prolog). It recasts the information in the standard `mdt` format and forwards the information to standard output where it can be read by the `mdt` process. Communication with Prolog is accomplished through the file system. This process also relays its standard input to standard output, permitting it to be part of a pipeline on the front end of the parser kernel.

`FromVerbex` This process listens for speech which has been interpreted by the commercial speech recognizer. The process translates the speech recognizers output into a

form which can be understood by `mdt` and then relays that information to its standard output where it can be read by the `mdt` process. This process also relays its standard input to standard output, permitting it to be part of a pipeline on the front end of the parser kernel.

`gap` This process accepts as input the dialog expectation and speech information which would normally be sent directly to the parser kernel, and modifies that information by adding arcs to the word lattice output from the speech recognizer. This is done in order to make up for the fact that the speech recognizer outputs only its best guess instead of a full lattice of word possibilities. The output of this process is a copy of its input with the extra input arcs added. Addition details concerning the operation of this process are found in subsection 6.1.3.

`ToDectalk` This filter intercepts the text of phrases which are to be spoken by the system and sends that text to the commercial speech synthesizer. The `ToDectalk` filter is used by the verification subsystem to utter the verification prompt. All inputs which are not text phrases to be spoken are passed to standard output unaltered.

`ToProlog` This program reformats the output of `mdt` and transmits the result to the dialog controller running under Prolog. The file system is used to communicate with Prolog.

The principal concern of this chapter, and indeed this whole thesis, is the parser kernel in the program `mdt`. The other processes, with the possible exception of `gap`, are not expected to be especially useful to other projects. They are included in this discussion in order to demonstrate how a collection of simple filters on the input and output of `mdt` can be used to easily interface the parser with the rest of an SLS without having to make changes to the parser kernel itself.

The parser subsystem is started in UNIX by a command such as this:

```
FromVerbex | FromProlog | ToDectalk | gap | mdt | ToProlog
```

This command invokes all processes and causes them to communicate with each other through pipes. Note that the above is a simplified version of the command which normally starts the parser running. On some of the processes in the pipeline there are often command line switches which are omitted here for brevity. It is also common to preface one or more processes with an `rsh` command to cause that process to run on a remote machine. In this way, the work of the parser can be spread among several computers. Spreading the work around is advantageous because both the parser kernel in `mdt` and the dialog controller running in Prolog use large amounts of memory and virtual memory thrashing may occur if both processes are run on the same machine. An `rsh` prefix must also be added to `FromVerbex` and `ToDectalk` if the speech recognizer and speech synthesizer are connected to machines other than the host.

6.1.1 `mdt`

The Minimum Distance Translator (mdt) uses the MDSDT algorithm (see section 3.3) together with dialog expectation matching (see section 3.4.1) to assign a translation to a lattice of word possibilities. The input word lattice is typically the output of a speech recognizer. The output is intended to be a statement in logic which embodies the meaning of what was spoken.

The translation from input to output is governed by a translation grammar. The translation grammar is read from a file when the program is initially invoked. The format of the translation grammar is described in subsection 6.1.2.

The Minimum Distance Translator is designed to be used as an interface between a speech recognizer and a natural language processing system. To this end, the operation of the translator has been kept as general as possible, so that it may be used in a wide variety of systems. It is envisioned that the communication between the translator and other parts of the system would occur through UNIX pipes, and that simple format conversion filters on the input and output of the translator would make necessary changes in the format of the data.

Command-line Options

- `cutoff=`*cost*

  The maximum translation cost per node in the input graph is set to the floating point value *cost*. The default is 1.5.

- `delcost=`*cost*

  The distance penalty associated with skipping or ignoring a word in the input lattice is set to the floating point value *cost*.

- `gram=`*filename*

  This causes the translation grammar to be read from the file named *filename*. If no file is specified by this option, then the grammar is read from a file named "mdtgram" in the current directory.

- `log=`*filename*

  This specifies the name of a file onto which a log of all translator activities is to be appended. If no log file is specified, then translator activities are not recorded anywhere.

- `-monitor`

  Causes the translator to print a summary of its activity on standard error. This is useful for real-time monitoring of what the translator is doing.

- `-noecho`

  The default action of the translator is to echo all its inputs to standard output. This switch suppresses the echo.

- `start=`*nonterminal*

  The distinguished nonterminal of the translation grammar (that is, the start symbol of the grammar) is set to be *nonterminal*. In the absence of this switch, the distinguished nonterminal is `StartState`.

- `timeout=`*number*

  Specify the maximum time (in seconds) which the translator will spend trying to translate a single input. If more time than this is required then the logical form verb—timeout— is output with an utterance cost of zero.

- `-tree`

  This switch will cause the translator to output not a logical form but a parse tree of the recognized input. This is useful for debugging of the translation grammar.

Input format

The Minimum Distance Translator takes all its input from standard input. The input is line oriented. No input prompt is generated. Lines which the translator does not know how to interpret are ignored. The translator understands and acts upon lines of the following form:

- `A` *from to cost label*

  Each line of this form describes an arc in the input word lattice. *From* and *to* are integers node numbers which specify the beginning and end of the arc, respectively. *Cost* is a real number matching cost for the arc, and *label* is the name of a terminal symbol in the input language of the translation grammar with which the arc should be labeled.

- `B`

  This input causes mdt to begin translation using the input graph and expectation previously entered. When translation is completed, each hypothesis in the hypothesis set is printed together with its utterance cost. The minimum matching string of the final result is also printed. After translation is complete, the expectation set and input graph are erased.

- `E` *cost expectation-string*

  The *expectation-string* is a sequence of characters which is interpreted as an hypothesis of the dialog expectation. The distance associated with that hypothesis is specified by the real number *cost*. The expectation string uses the asterisk as a wildcard character.

- `T` *word-list*

  This is a short-cut method for manually entering a universal expectation and a linear graph as a test input to the translator. The translator will echo the sequence of `A`, `B`, and `E` commands which will have the same effect as this short-cut.

Output format

Every input line is echoed to standard output regardless of whether or not the translator understood the line or acted upon it. When a `T` line is input the corresponding sequence of `A`, `B`, and `E` commands is output as well. When translation occurs the hypothesis set is output by a sequence of lines described below.

- `H` *utterance-cost hypothesis-string*

  Each hypothesis generated by the MDSDT algorithm is output in a line of this form. The current implementation uses the N-ties method of recording hypotheses, so the utterance cost will be the same for every hypothesis.

- `P` *total-cost hypothesis-string*

  The hypothesis which matches the lowest-cost expectation string is output in a line of this form. The hypothesis string on a `P` line has been matched against an expectation and therefore may have had some of its wildcards expanded.

6.1.2 Format of the Translation Grammar

This subsection describes the file format of the translation grammar.

The translation grammar consists of a unordered sequence of production rules. Whitespace is ignored, except where it delimits tokens. Comments are also ignored and can occur anywhere within the translation grammar file. Comments are as in C (beginning with "`/*`" and ending with "`*/`") or as in C++ (beginning with "`//`" and ending at the next newline character). Comments do not nest.

Each production rule begins with a nonterminal symbol. A nonterminal may be any sequence of upper or lower case letters, digits, or the underscore symbol "`_`" except that the first character may not be a digit. This first nonterminal is the left-hand side (LHS) of the production rule. Following the LHS is the reserved token "`->`". Next is a sequence of zero or more nonterminal and input terminal symbols which comprise the right-hand side (RHS) of the production rule. Terminal symbols alphanumeric identifiers like nonterminals, or they may be numeric constants, or they may be any punctuation character which does not have a special meaning in the production rule. The following symbols have special meaning:

```
->  :  '  ?  !  {  }  .
```

The period may be used as a terminal symbol if it is preceded by a backslash, like this: "`\.`".

The reserved token "`:`" closes the RHS of the rule and begins the semantics. The semantics consists of a sequence of nonterminal and output terminal symbols. The semantics section and the entire production rule is terminated by a period. If a production rule has no symbols in the semantics then the colon which normally precedes the semantics may be omitted.

The mapping from semantic nonterminals to RHS nonterminals is indicated through the use of apostrophes following corresponding nonterminal symbols. If a certain nonterminal in the semantics maps to a certain nonterminal in the RHS of the rule then both nonterminals should be followed by the same number of apostrophes. If the same nonterminal occurs more than once in the semantics then each occurrence must be followed by a unique number of apostrophes.

Insertion costs for symbols on the RHS of a production rule are indicated by special marks following each symbol. If the symbol is not followed by any special mark then its insertion cost is 1.0 if it is a terminal. If the symbol is a nonterminal then the default insert cost is computed by finding the minimum total insertion cost of every rule for which that symbol is the LHS. If any symbol if followed by a question mark then the insertion cost of that symbol is taken to be zero. This effectively makes the symbol optional in the input of the parser. If a RHS symbol is followed by an exclamation point the insertion cost for that symbol is raised to infinity thereby making it impossible for that symbol to be inserted. Any other insertion cost may be specified by following the symbol with a floating point number enclosed in curly braces, like this: "`{4.2}`".

Example translation grammars

The following text is an example of how the simple translation grammar shown on page 32 would be coded for use by the `mdt` program.

```
S -> C S' S'' : S' S'' C.
S -> B S : S B.
S -> A : A.
A -> a : a.
A -> b : b.
⋮
```

```
A -> z : z.
B -> - : -.
C -> + : +.
C -> $\times$ : $\times$.
```

The next example is 12 rules (out of 505) from the grammar used in the experiments. The 12 rules recognize all utterances which mean the same as "the LED is displaying a one and a seven". This set of rules is not intended to typify a good grammar design, or to serve as a template for future grammar development. The rules are shown purely to demonstrate the syntax of the grammar specification. The nonterminal `StartState` is the initial nonterminal of the grammar.

```
StartState -> A' : A'.
A -> L' DS' : assertion(true,state(display,L',DS')).
A -> L' displaying DS' :
     assertion(true,state(display,L',DS')).
A -> there is DS' :
     assertion(true,state(display,led,DS')).
DS -> OS' : OS'.
L -> the? led : led.
L -> the? light : led.
L -> it : *.
OS -> one seven : [eight,t,u,u,u,t,u,u,u,u,u,u,u].
OS -> seven one : [eight,t,u,u,u,t,u,u,u,u,u,u,u].
OS -> one and seven : [eight,t,u,u,u,t,u,u,u,u,u,u,u].
OS -> seven and one : [eight,t,u,u,u,t,u,u,u,u,u,u,u].
```

6.1.3 gap

The graph augmenting preprocessor (`gap`) program accepts as input the dialog expectation and word lattice information which would normally be sent directly to the parser kernel and modifies that information by adding new arcs to the word lattice. The arcs are added in order to make up for the fact that the speech recognizer outputs only its best guess instead of a full lattice of word possibilities. The output of this process is a copy of its input with the extra input arcs added.

The `gap` program accepts the following command line switches:

- `dict=`*filename*

  The name of a dictionary file is specified by *filename*. If no dictionary file is specified in this way, then the dictionary is read from a file named `mdtdict`.

- `scale=`*value*

  The penalty at the beginning of each entry in the dictionary is multiplied by the amount *value*. If no scaling value is specified then *value* is assumed to be 1.0.

The program adds additional arcs to the input graph in parallel with existing arcs. The *dictionary file* specifies what arcs are added. The dictionary file consists of lines containing a penalty value followed by two words. When ever an arc labeled by the first word is seen, then a new arc is created which is labeled by the second word and the distance of the new arc is the distance of the original arc plus the amount of the penalty.

An example

Suppose the dictionary file contains the following entries:

```
0.18     a                    i
0.03     a                    and
0.44     a                    only
0.973    a                    the
0.11     and                  a
0.02     and                  an
0.48     one                  on
1.2      one                  point
0        one                  want
0.24     one                  what
1.76     seven                nothing
```

Then let the input to the `gap` program be the following input graph.

```
A  0  1      0.1 a
A  1  2     0.02 one
A  2  3     0.34 and
A  3  4     1.23 a
A  4  5     0.06 seven
```

Then, the output of the `gap` program would be the augmented input graph shown below.

```
A  0  1      0.1 a
A  0  1     0.13 and
A  0  1     0.28 i
A  0  1     0.54 only
A  0  1     1.07 the
A  1  2     0.02 one
A  1  2      0.5 on
A  1  2     1.22 point
A  1  2     0.02 want
A  1  2     0.26 what
A  2  3     0.34 and
A  2  3     0.45 a
A  2  3     0.36 an
A  3  4     1.23 a
A  3  4     1.26 and
A  3  4     1.41 i
A  3  4     1.67 only
A  3  4      2.2 the
A  4  5     0.06 seven
A  4  5     1.82 nothing
```

6.2 The Grammar Editor

6.2.1 `ged`

The core of the grammar editor is the program `ged`. The grammar editor is interactive, and so it is normally invoked without any arguments. However, one can specify a default grammar start symbol with an argument of the form `start=`*symbol*. When this switch is omitted, the default start symbol is `StartState`.

When `ged` is started, the program prints the prompt

```
ged =>
```

then halts waiting for the user to type a command. To end the editing session, the user types control-D at the command prompt.

Commands

Commands to the grammar editor can take any of the following five forms:

- `help`

  This is the easiest but perhaps the most important command in the command repertory. In response to this command, the grammar editor prints a summary of all the other commands that it understands.

- *identifier* `=` *expression*

  A new grammar subset is created using the assignment command. The identifier is a alphanumeric name which uniquely identifies the new subset. The expression is a description of how the subset is generated. Section 6.2.1 describes the syntax of an expression.

  If the identifier in an assignment command has been previously used to mark an earlier subset then the earlier subset is deleted after the new is computed. This allows the identifier on the left to be used as part of the expression on the right. For example, to merge the contents of grammar subset `X` into grammar subset `Y`, it suffices to type:

  ```
  Y = Y + X
  ```

  If the expression is omitted from an assignment command then the subset referred to by the identifier is deleted.

- `show` *type* `in` *expression*

  This command causes features of a grammar subset to be listed. The initial word in the command can be changed to `count` and the same feature will be counted instead. The *expression* describes the grammar subset from which the features are read.

  The *type* field in the command specifies what feature of the grammar subset is displayed. Acceptable values for *type* are:

  - `rules`

    This causes production rules of the grammar subset to be listed.

  - `terminals`

    This causes all input terminal symbols used the grammar to be listed.

- `subsets`

  This type lists all grammar subsets which have been given names in the grammar editor. The "`in` *expression*" part of the command should be omitted when this type is used.

- `variables`

  This causes all nonterminal symbols in the grammar subset to be listed.

- `inputs from` *nonterminal*

  This causes all well-formed inputs to the grammar subset to be listed in lexical order. The argument *nonterminal* specifies which nonterminal of the grammar subset to use as the root of the parse tree. The "`from` *nonterminal*" may be omitted from this type and the default grammar start symbol will be used. The program `leg` (see section 6.2.2) to compute this type.

- `outputs from` *nonterminal*

  This causes all well-formed outputs from the grammar subset to be listed in lexical order. The "`from` *nonterminal*" extension works the same as with the `input` type. The program `leg` (see section 6.2.2) to compute this type.

- `parse` *text* `from` *nonterminal* `in` *expression*

  This command causes the linear input *text* to be parsed using the MDSDT algorithm and the grammar subset specified by *expression*. The root of the parser tree is *nonterminal*, except if the "`from` *nonterminal*" portion of the command is omitted the default start symbol is used. The *text* should be enclosed in double quotes, or in balanced square brackets.

  If the initial word of this command is changed to `tree` a parse tree is generated instead of a standard parse.

  This command is implemented as a call to the program `mdt` which is described in sectionc5-mdt.

- `write` *expression* `to` *filename*

  This command causes a grammar subset to be written to a file. The subset to be written is specified by *expression* and the filename is given by *filename*. The filename should be delimited by balanced square brackets or by double quotes if it contains any special characters (such as a period.)

Expressions

An important part of the grammar editors command syntax is the flexible method used to describe a grammar subset. The following bullets show how a grammar subset is constructed. Each of the following bullets constitutes an *expression* in the command level syntax of the previous subsection.

- `read` *filename*

  An expression of this form causes the grammar to be read from a file specified by *filename*. The filename should be delimited by matched square brackets or by double quotes.

- *identifier*

  Any of the named grammar subsets created by an assignment statement can be used as an expression simply by calling its name.

- `(` *expression* `)`

  Parenthesis can be used in expressions to show grouping.

- *expression* `+` *expressions*

  An expression of this form specifies a grammar which is the union of all rules in the two subexpressions.

- *expression* `-` *expressions*

  The grammar specified here is the set of all production rules in the first subexpression which do not appear in the grammar of the second subexpression.

- *expression* `&` *expressions*

  This operation specifies the intersection of the subexpressions.

- `match` *pattern* `in` *expression*

  This expression specifies a grammar subset which is all rules in the grammar of the subexpression which match the *pattern*. The UNIX utility `grep` is used for pattern matching. The pattern should be delimited by matching square brackets or by double quotes.

- `parents of` *expression1* `in` *expression2*

  In this, *expression1* must be a subset of *expression2*. The resulting grammar subset is all production rules in *expression2* which can derive production rules in *expression1*. The production rules of *expression1* are not included in the result of this expression.

- `children of` *expression1* `in` *expression2*

  This is the counterpart to the `parents` expression in the previous bullet. The resulting grammar subset is all production rules in *expression2* which can be derived from production rules in *expression1*. The production rules of *expression1* are not included in the result of this expression.

- `generate` *text* `from` *nonterminal* `in` *expression*

  The resulting grammar is the subset of production rules in *expression* which can be used to generate the semantic string *text* when the parse tree is rooted at *nonterminal*. The "`from` *nonterminal*" subphrase can be omitted and the default start symbol will be used. The *text* field should be delimited by either matching square brackets or double quotes.

  Text can be a regular expression but it must not specify an null string. The regular expression operators are "`\*`" Kleene closure, "`\|`" for alternation, and "`\(`" and "`\)`" for grouping. The special symbol "`\.`" will match any token. Thus the regular expression "`\.\.*`" will match any phrase of one or more symbols.

  This subset is computed using the program `rg` which is described in section 6.2.3.

- `recognize` *text* `from` *nonterminal* `in` *expression*

  This is the counterpart of the `generate` expression of the previous bullet. The resulting grammar is the subset of production rules in *expression* which can be used to recognize the input string *text* when the parse tree is rooted at *nonterminal*. *Text* can be a regular expression. This subset is computed using the program `rg` which is described in section 6.2.3.

- `fluff of` *expression1* `in` *expression2*

  This seemingly strange operator is designed for use in conjunction with the `generates` operator. The resulting grammar is the the set of production rules in *expression2* which have RHSs which are terminal symbols in the grammar of *expression1*. Children, grandchildren, and so forth of the designated production rules are also included in the result.

- `edit` *expression*

  To evaluate this expression, the grammar editor first evaluates the subexpression, then calls the user's standard editor (specified in the `VISUAL` or `EDITOR` environment variables of UNIX) to make changes to that grammar. The result of the expression is the grammar after the editing session.

Examples

Some examples of typical `ged` command lines and their meanings will help to elucidate the above material. All of the examples in this section are printed in the typewriter font and no distinction is made between what the program prints and what the user types. Presumably the distinction is obvious.

The first operation after invoking the grammar editor is typically something like the following:

```
ged => all = read "mdtgram"
```

This command creates a new subset named `all` and fills it with the production rules read from the file `mdtgram`. (Recall from section 6.1.1 that `mdtgram` is the default name of the grammar used by the `mdt` parsing program.)

Now suppose that the user wants to edit all production rules having to do with the input terminal symbol "switch". One might begin this way:

```
ged => b = match "switch" in all
```

This command creates a new grammar subset named `b` which is all production rules in `all` matching the pattern `switch`. Now the user checks to see how many rules are in `b` and what those rules are:

```
ged => count rules in b
2
ged => show rules in b
SW -> switch : switch.
SW -> power switch : switch.
```

Next the user creates a new variable which contains both the rules of `b` and all of the parents of those rules:

```
ged => b2 = b + parents of b in all
```

The user now enters a complex command which causes the rules in `b2` to be edited and then reinserted into the full grammar.

```
all = (all-b2)+edit b2
```

After the grammar is the way the user wants it, it can be written to a new file using the following command:

```
write all to "new.mdtgram"
```

6.2.2 `leg`

The part of the grammar editor which lists and counts the number of input or output strings in a grammar is a separate program called `leg`. (The name is an acronym for List Elements of a Grammar.) This program implements algorithm 4.1.

The program is invoked by typing its name followed by any optional switches. The program then generates strings in the language of the grammar in lexical order. Valid switches to this program are as follows:

- `from=`*integer*

  This switch specifies the length in tokens of the shortest string printed. The default value is 1.

- `gram=`*filename*

  This specifies the name of the file from which the grammar is read. If this switch is omitted the file `mdtgram` is used.

- `nx=`*nonterminal*

  This switch forces the nonterminal symbol *nonterminal* to be treated as if it were a terminal symbol. In other words, rules in the grammar which have *nonterminal* as their LHS are not used.

- `side=`*integer*

  This switch is used to control whether the strings generated are from the input language or the output language of the input grammar. If the value of *integer* is 0 then the input language is used but output language strings are printed if *integer* is 1.

- `start=`*nonterminal*

  This switch specifies the start symbol of the grammar – the nonterminal which will be the root of every parser tree. If the switch is omitted, the nonterminal `StartState` is assumed.

- `to=`*integer*

  This switch specifies the length of the longest string which will be printed by the program. The default value is 1000.

6.2.3 `rg`

The `generate` and `recognize` operators in expressions of the grammar editor are computed using the program `rg`. (The name `rg` is derived from the first letters of "generate" and "recognize".) When invoked, the `rg` program first read a grammar, then begins reading lines from standard input. Each line of input is assumed to contain a single regular expression. For each regular expression, the subset of production rules in the grammar which might be used to parse any string in the language of that regular expression is printed on standard output. The program terminates when it reaches the end of its input. The algorithm described in section 4.2 is used to find the appropriate production rules.

Command line switches can be used to alter the behavior of the program. The following command line switches are recognized:

- `gram=`*filename*

  The name of the file from which the grammar is read is specified by this switch. If the switch is omitted, then the default file name `mdtgram` is used.

- `side=`*filename*

  This switch determines whether the regular expressions describe strings in the input language or the output language of the grammar. If the value of *integer* is 0 then the input language is intended but output language strings are used if *integer* is 1.

- `start=`*nonterminal*

  This switch specifies the start symbol of the grammar – the nonterminal which will be the root of every parser tree. If the switch is omitted, the nonterminal `StartState` is assumed.

The regular expression notation is similar to what is commonly seen except that all operators are prefaced by a backslash character in order to distinguish them from ordinary tokens. The symbol `\*` following any regular expression means that there may be zero or more repetitions of that expression. The symbol `\|` between two regular expressions means that either one or the other of the expressions may be used. The two symbols `\(` and `\)` are used for grouping. Finally, the special symbol `\.` means any ordinary token. Hence the idiom `\.\*` means zero or more tokens of any kind.

6.3 The Verification Subsystem

The verification subsystem is implemented as in Prolog as a preprocessor to the dialog controller. The subsystem can not be used independently and unlike the parser is not portable to other SLSs. No interesting information can be discerned from a detailed description of the implementation that can not as easily be gathered from the material in chapter 5. For that reason, a full description of the verification subsystem implementation is omitted.

Chapter 7

Conclusions

7.1 Summary of New Results

The major results described in the preceding chapters are the following:

- A new kind of parser suitable for spoken natural language dialog is described. A new and faster algorithm for minimum-distance parsing forms the core of the new parser, but syntax-directed translation and the use of dialog expectation also play important roles. In live experiments using the new parser, 83% of user inputs were correctly interpreted even though 43% of these inputs were grammatically ill-formed. Followup analysis of the data collected during these experiments showed that dialog expectation was only helpful in breaking ties between competing meanings found by the minimum-distance parsing algorithm. This last result is surprising – it was anticipated that dialog expectation would be much more useful in determining the correct parse.

- A development tool useful in the construction of large natural language grammars is described, and algorithms used by this development tool are analyzed. The grammar development system described is unique in that it combines into a simple and consistent interface the most complete collection of features of any such previously reported system. In fact, some of the functions of the grammar development system are original and have never been previously described in the literature. New algorithms are presented for listing and counting the number of elements in a context-free language, and for finding small subsets of the grammar's production rules which are relevant to a particular class of meanings or inputs. The problem of counting the number of elements in a context-free language is shown to be NP-hard.

- A subsystem for selectively verifying the deduced meaning of an utterance when that meaning is in doubt is described. When added to a complete dialog processing system, the verification subsystem improves the percentage of correct parses from 83% to 97% without excessive additional dialog or burden upon the user. The idea of selectively verifying the deduced meaning of an utterance in order to improve the effective accuracy of the parser is original. An extensive search of the technical literature, and discussions with established computational linguistics researchers revealed no prior mention of this idea.

- Finally, an overview of the actual implementations of the above ideas was given. This overview contributes to the collective knowledge of computer science by providing a proven example of an effective method for implementing similar systems, and by providing a "user's reference

manual" to other researchers who may choose to use or extend parts of this system in future studies.

7.2 Future Directions

This thesis describes working, practical solutions to several problems associated with the design and development of spoken natural language dialog parsing systems. Yet, as is typical for an investigative study, the results obtained pose as many questions as they answer, and therefore suggest topics for future study. Among the questions for which further investigation is desired are the following:

- Does the new parsing strategy work as well in other problem domains as it does with repair dialogs? How well would the MDSDT parsing technique work in an automated airline reservation system, for example?
- Is dialog expectation really as weak as the above results suggest? There seems to be a common belief among computational linguistics researchers that the more access a parser has to high-level dialog expectation, the better. The findings of this thesis call that belief into question. Collaborative studies will be necessary in order to convince most researchers that dialog expectation is really as weak as has been reported here.
- How easily does the system described here scale up? What would be the effect of increasing the vocabulary size by a factor of 10, say? Would dialog expectation become more important if the scope of the problem domain were increased? Speculation abounds about these and similar questions, but there are, so far, no solid answers.
- Speech recognition errors were a significant cause of parsing errors. In what ways might it be possible for the parser to help the speech recognizer provide more accurate estimates of the user's words?
- How much of the developer's time can the grammar development system really save? What fraction of the total system development time will be spent writing the grammar if the grammar development tools described above are employed?
- What are some reasonable techniques for dynamically computing the verification threshold in the selective verification subsystem so that utterances of lesser importance are not as frequently verified? In the implementation, the verification threshold was set very high for questions and at a much lower value for all other inputs. Surely there must be a better way to chose the verification threshold than this simple binary heuristic.

The above list of questions is, of course, incomplete. Much work remains to be done before the design of practical and robust natural language dialog parsing systems becomes routine. Even so, it is the opinion of this author that such a goal is ultimately achievable and will yield to the continuing efforts of researchers.

Bibliography

[1] A. V. Aho and J. D. Ullman. Properties of syntax directed translations. *Journal of Computer and System Sciences*, 3(3):319–334, 1969.

[2] Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computation*, 1(4):305–312, 1972.

[3] J. F. Allen. *Natural Language Understanding*. The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1987.

[4] Kenneth R. Beesley and David Hefner. PeriPhrase: Lingware for parsing and structural transfer. In *COLING-86: Proceedings of the 11th International Conference on Computational Linguistics*, pages 390–392, Bonn, August 1986.

[5] Bran Boguraeu, John Carroll, Ted Briscoe, and Claire Grover. Software support for practical grammar development. In *COLING-88: Proceedings of the 12th International Conference on Computational Linguistics*, pages 54–58, Bonn, August 1986.

[6] A. Chapanis. Interactive human communication: Some lessons learned from laboratory experiments. In B. Shackil, editor, *Man-Computer Interaction: Human Factors Aspects of Computers and People*, pages 65–114. Sijthoff and Noordhoff, Rockville Md., 1981.

[7] Y. L. Chow, M. O. Dunham, O. A. Kimball, M. A. Krasner, G. F. Kubala, J. Makhoul, P. J. Price, S Roucos, and R. M. Schwartz. BYBLOS: the BBN continuous speech recognition system. In Alex Waibel and Kai-Fu Lee, editors, *Readings in speech Recognition*, pages 596–599. Morgan Kaufman, San Mateo, CA, 1990.

[8] Jay Earley. An efficient context-free parsing algorithm. *Communcations of the ACM*, 13(2):94–102, 1970.

[9] C. M. Eastman and D. S. McLean. On the need for parsing ill-formed input. *American Journal of Computational Linguistics*, 7(4):257, 1981.

[10] Linda Fineman. Questioning the need for parsing ill-formed inputs. *American Journal of Computational Linguistics*, 9(1):22, 1983.

[11] P. E. Fink and A. W. Biermann. The correction of ill-formed input using history based expectation with application to speech understanding. *American Journal of Computational Linguistics*, 12(1):13–36, 1986.

[12] R. E. Frederking. *Integrated Natural Language Dialogue: A Computational Model*. Kluwer Academic Publishers, Boston, 1988.

[13] Egidio P. Giachin and Claudio Rullent. Robust parsing of severely corrupted spoken utterences. In *COLING-88: Proceedings of the 12th International Conference on Computational Linguistics*, pages 196–201, 1988.

[14] R. Guindon, K. Shuldberg, and J. Conner. Grammatical and ungrammatical structures in user-advisor dialogues: Evidence for sufficiency of restricted languages in natural language interfaces to advisory systems. In *ACL Proceedings, 25th Annual Meeting*, Stanford, CA, 1987.

[15] Hans Haugeneder and Manfred Gehrke. A user friendly ATN programming environment (APE). In *COLING-86: Proceedings of the 11th International Conference on Computational Linguistics*, pages 399–401, Bonn, August 1986.

[16] Philip J. Hayes, Alexander G Hauptmann, Jaime G. Carbonell, and Masaru Tomita. Parsing spoken language: A semantic caseframe approach. In *COLING-86: Proceedings of the 11th International Conference on Computational Linguistics*, pages 587–592, Bonn, August 1986.

[17] Philip J. Hayes and George V. Mourandian. Flexible parsing. *American Journal of Computational Linguistics*, 7(4):232–242, 1981.

[18] D. R. Hipp and R. W. Smith. A demonstration of the "circuit fix-it shoppe". A 12 minute videotape available from the authors at Duke University, Durham, NC 27706, August 1991.

[19] John E. Hopcroft and Jeffrey D.Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 1979.

[20] Ajay N. Jain and Alex H. Waibel. Robust connectionist parsing of spoken language. In *ICASSP-91*, pages 593–596, 1991.

[21] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[22] Lauri Karttunen. D-PATR: a development environment for unification-based grammars. In *COLING-86: Proceedings of the 11th International Conference on Computational Linguistics*, pages 74–80, Bonn, August 1986.

[23] Kenji Kita and Wayne H. Ward. Incorporating lr parsing into SPHINX. In *ICASSP-91*, pages 269–272, 1991.

[24] Bernard Laug. Parsing incomplete sentences. In *COLING-88: Proceedings of the 12th International Conference on Computational Linguistics*, pages 365–371, 1988.

[25] Kai-Fu Lee, Hsiao-Wuen Hon, and Raj Reddy. An overview of the SPHINX speech recognition system. In Alex Waibel and Kai-Fu Lee, editors, *Readings in speech Recognition*, pages 600–610. Morgan Kaufman, San Mateo, CA, 1990.

[26] Stephen E. Levinson. Structural methods in automatic speech recognition. *Proceeding of the IEEE*, 73(11):1625–1650, 1985.

[27] Gordon Lyon. Syntax-directed least errors analysis for context-free languages. *Communcations of the ACM*, 17(1):3–14, 1974.

[28] D. Memmi and J. Mariani. ARBUS: A tool for developing application grammars. In *COLING-82*, pages 221–226, 1982.

[29] Herman Ney. Dynamic programming parsing for context-free grammars in continuous speech recognition. *IEEE Transactions on Signal Processing*, 39(2):336–340, 1991.

[30] Nils J. Nilsson. *Principles of Artifical Intelligence*. Morgan Kaufmann Publishers, Inc., Los Altos, California, 1980.

[31] M. Okada. A unification-grammar-directed one-pass search algorithm for parsing spoken language. In *ICASSP-91*, pages 721–724, 1991.

[32] Annedore Paeseler and Herman Ney. Continuous-speech recognition using a stochastic language model. In *ICASSP-89*, pages 719–722, 1989.

[33] M. J. Russell, K. M. Ponting, S. M. Peeling, S. R. Browning, J. S. Bridle, and R. K. Moore. The ARM continuous speech recognition system. In *ICASSP-90*, pages 69–72, 1990.

[34] George W. Smith. *Compters and Human Language*. Oxford University Press, Oxford, 1991.

[35] Ronnie W. Smith. *A Computational Model of Expectation-Driven Mixed-Initiative Dialog Processing*. PhD thesis, Duke University, September 1991.

[36] Ronnie W. Smith, D. Richard Hipp, and Alan W. Biermann. A dialog control algorithm and its performance. Unpublished as of this writing, but has been submitted to the 3rd Conference on Applied Natural Language Processing., 1992.

[37] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the Association for Computing Machinery*, 21:168–173, 1974.

[38] Ye-Yi Wang and Alex Waibel. A connectionist model for dialog processing. In *ICASSP-91*, pages 785–788, 1991.

[39] W. H. Ward, A. G. Hauptman, R. M. Stern, and T. Chanak. Parsing spoken phrases despite missing words. In *ICASSP-88*, volume 1, pages 275–278, 1988.

[40] Wayne Ward. Understanding spontaneous speech: The phoenix system. In *ICASSP-91*, pages 365–367, 1991.

[41] J. White. The research environment in the METAL project. In S. Nirenburg, editor, *Machine Translation: Theoretical and Methodological Issues*. Cambridge University Press, 1987.

[42] S. J. Young, N. H. Russell, and J. H. S. Thornton. The use of syntax and multiple alternatives in the VODIS voice operated database inquiry system. *Computer Speech and Language*, 5(1):63–80, 1991.

[43] Sheryl R. Young, Alexander G. Hauptmann, Wayne H. Wood, Edward T. Smith, and Philip Warner. High level knowledge sources in usable speech recognition systems. In Alex Waibel and Kai-Fu Lee, editors, *Readings in speech Recognition*, pages 538–549. Morgan Kaufman, San Mateo, CA, 1990.

[44] D. M. Younger. Recognition and parsing of context free languages in time n^3. *Information and Control*, 10, 1967.

[45] V. Zue, J. Glass, D. Goodine, H. Leung, M. Phillips, J. Polifroni, and S. Seneff. Integration of speech recognition and natural language processing in the MIT voyager system. In *ICASSP-91*, pages 713–716, 1991.

[46] Victor Zue, James Glass, David Goodine, Hong Leung, Michael Phillips, Joseph Polifroni, and Stephanie Seneff. The VOYAGER speech understanding system: Preliminary development and evaluation. In *ICASSP-90*, pages 73–76, 1991.

[47] Victor Zue, James Glass, David Goodine, Michael Phillips, and Stephanie Seneff. The SUMMIT speech recognition system: Phonological modelling and lexical access. In *ICASSP-90*, pages 49–52, 1990.

Biography

Dwayne Richard Hipp was born to Richard E. and Dorothy Springs Hipp on April 9, 1961 in Charlotte, North Carolina. He grew up in Stone Mountain, Georgia, and graduated from Stone Mountain High School in 1979. He attended the Georgia Institute of Technology in Atlanta beginning in the fall of 1979. The degree Bachelor of Electrical Engineering was awarded him with highest honor in 1983, and he received a Master of Science in Electrical Engineering, also from Georgia Tech, the following year. Between 1984 and 1987, he worked as an Engineer at AT&T Technologies in Greensboro, North Carolina, but then left to pursue a Ph.D. in Computer Science at Duke University beginning in the fall of 1987.