GLR* :
A Robust Grammar-Focused Parser for Spontaneously Spoken Language

Alon Lavie

May 1996
CMU-CS-96-126

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee:
Masaru Tomita, Chair
Jaime Carbonell
Alex Waibel
Edward Gibson, MIT

Keywords: Natural Language Processing, Speech Understanding, Machine Translation, Parsing, Generalized LR Parsing, JANUS

Abstract

The analysis of spoken language is widely considered to be a more challenging task than the analysis of written text. All of the difficulties of written language can generally be found in spoken language as well. Parsing spontaneous speech must, however, also deal with problems such as speech disfluencies, the looser notion of grammaticality, and the lack of clearly marked sentence boundaries. The contamination of the input with errors of a speech recognizer can further exacerbate these problems. Most natural language parsing algorithms are designed to analyze "clean" grammatical input. Because they reject any input which is found to be ungrammatical in even the slightest way, such parsers are unsuitable for parsing spontaneous speech, where completely grammatical input is the exception more than the rule.

This thesis describes GLR*, a parsing system based on Tomita's Generalized LR parsing algorithm, that was designed to be robust to two particular types of extra-grammaticality: noise in the input, and limited grammar coverage. GLR* attempts to overcome these forms of extra-grammaticality by ignoring the unparsable words and fragments and conducting a search for the maximal subset of the original input that is covered by the grammar. The parser is coupled with a beam search heuristic, that limits the combinations of skipped words considered by the parser, and ensures that the parser will operate within feasible time and space bounds.

The developed parsing system includes several tools designed to address the difficulties of parsing spontaneous speech. To cope with high levels of ambiguity, we developed a statistical disambiguation module, in which probabilities are attached directly to the actions in the LR parsing table. The parser must also determine the "best" parse from among the different parsable subsets of an input. We thus designed a general framework for combining a collection of parse evaluation measures into an integrated heuristic for evaluating and ranking the parses produced by the GLR* parser. This framework was applied to a set of four parse scoring measures developed for the JANUS scheduling domain and the ATIS domain. We added a parse quality heuristic, that allows the parser to self-judge the quality of the parse chosen as best, and to detect cases in which important information is likely to have been skipped.

To demonstrate its suitability to parsing spontaneous speech, the GLR* parser was integrated into the JANUS speech translation system. Our evaluations on both transcribed and speech recognized input have indicated that the version of the system that uses GLR* produces between 15% and 30% more acceptable translations, than a corresponding version that uses the original non-robust GLR parser. We also developed a version of GLR* that is suitable to parsing word lattices produced by the speech recognizer, and investigated how lattice parsing can potentially overcome errors of the speech recognizer and further improve end-to-end performance of the speech translation system.

Acknowledgements

There are many who have helped me along the long road that has culminated in this thesis, some in direct and obvious ways, others in small and supposedly unrelated ways, but important nonetheless.

I would like to thank my advisor, Masaru Tomita, for inspiring and supporting my research interests in parsing algorithms, for introducing me to the problems of spoken language parsing and understanding, for suggesting the topic of this thesis, and for guiding me in my thesis work.

I also wish to thank the other members of my thesis committee: Jaime Carbonell, Alex Waibel and Ted Gibson. Alex brought me into the JANUS speech-to-speech translation project, which provided a natural and practical setting for applying and testing my work. I am particularly grateful for his guidance on system performance issues and evaluation methods. Ted Gibson provided me with objective insight about my work, and with careful and well thought comments on my thesis draft. Special thanks are due to Jaime Carbonell, for sharing his experienced perspective on Machine Translation and AI, for stepping into the "advisor shoes" in Tommy's absence, and for his very helpful comments and suggestions on the preliminary draft of this thesis.

I would like to thank all my friends and colleagues in the JANUS project, for providing a fun and exciting environment for conducting research on speech understanding and translation. Particular thanks go to Lori Levin, Donna Gates, Oren Glickman, Noah Coccaro, Carolyn Rosé, Marsal Gavaldà, Laura Mayfield, Keiko Horiguchi and Kaori Shima, who worked closely with me on the project, and assisted me in a variety of experiments and evaluations reported in this thesis.

On the personal side, there is a whole bunch of friends, colleagues and family members, whom I would like to thank for their support and encouragement. I will not even attempt to list them all, for fear that surely someone will be forgotten. Yet, I feel a need to mention a handful of people to whom special thanks are due:

To my friend and ex-officemate Shai, who greeted and hosted me when I first arrived here in Pittsburgh, spent five years with me in a windowless Wean office, and helped me with so many "system hacking" questions...

To my best friend and next door neighbor Dean, who is always there to listen and give advice, and is really good at putting things in perspective.

To my good friend Orna, for being just that, but also for the numerous dinners and coffee breaks during the last two intense months of writing, that made them so much more bearable.

To my family in Israel, for their support, and in particular to my father, who not only provided constant encouragement and advice, but has also been a true role model for me to follow.

And most of all, to Bob, for taking care of "things" during the very busy months of final thesis writing, and for traveling with me along the longest and most difficult segments of the road to the PhD. I couldn't have done it without you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Overview

1.1. Introduction

The analysis of spoken language is widely considered to be a more challenging task than the analysis of written text. All of the difficulties of written language (such as ambiguity) can generally be found in spoken language as well. However beyond these difficulties, there are additional problems that are specific to the nature of spoken language in general, and spontaneous speech in particular. The major additional issues that come to play in parsing spontaneous speech are speech disfluencies, the looser notion of grammaticality that is characteristic of spoken language, and the lack of clearly marked sentence boundaries. The contamination of the input with errors of a speech recognizer can further exacerbate these problems.

Most natural language parsing algorithms are designed to analyze "clean" grammatical input. By the definition of their recognition process, these algorithms are designed to detect ungrammatical input at the earliest possible opportunity, and to reject any input that is found to be ungrammatical in even the slightest way. This property, which requires the parser to make a complete and absolute distinction between grammatical and ungrammatical input, makes such parsers fragile and of little value in many practical applications. Such parsers are thus unsuitable for parsing spontaneous speech, where completely grammatical input is the exception more than the rule.

This thesis describes GLR*, a parsing system based on Tomita's Generalized LR parsing algorithm, that was designed to be robust to two particular types of extra-grammaticality: noise in the input, and limited grammar coverage. GLR* attempts to overcome these forms of extra-grammaticality by ignoring the unparsable words and fragments and conducting a search for the maximal subset of the original input that is covered by the grammar.

Because the GLR* parsing algorithm is an enhancement to the standard GLR context-free parsing algorithm, grammars, lexicons and other tools developed for the standard GLR parser can be used without modification. GLR* uses the standard LR parsing tables that are compiled in advance from the grammar. It inherits the benefits of GLR in terms of ease of grammar development, and, to a large extent, efficiency properties of the parser itself. In the case that an input sentence is completely grammatical, GLR* will normally return the exact same parse as the standard GLR parser.

Although it should prove to be useful for other practical applications as well, GLR* was designed to be particularly suitable for parsing spontaneous speech. Grammars developed for spontaneous speech can concentrate on describing the structure of the meaningful clauses and sentences that are embedded in the spoken utterance. The GLR* parsing architecture can facilitate the extraction of these meaningful clauses from the utterance, while disregarding the surrounding disfluencies.

1.1.1. Extra-grammaticalities in Spontaneous Speech

Disfluencies in Spontaneous Speech

Because spontaneous speech is an "on-line" form of communication, it contains many kinds of disfluencies. The typical forms of disfluencies that can be found in spontaneous speech are false beginnings, repeated words, stutters, and various filled pauses (such as "ah","um" etc.). Disfluencies are speaker dependent, and vary greatly from one speaker to another. However, the parser should be able to disregard most such disfluencies, regardless of the speaker. Here are some examples, taken from transcriptions of spontaneous conversations in the scheduling domain, that demonstrate these phenomena. For each example we show the original utterance along with a "clean" version of the utterance, that contains just the meaningful content and omits the disfluencies. The "clean" versions correspond to what we would like the parser to detect and successfully parse.

- False beginning, filled pauses:
  - Original input: `um okay then yeah Monday at two is fine.`
  - Content input: `Monday at two is fine.`
- Repetition, filled pauses:
  - Original input: `uh I I need to meet with you next week.`
  - Content input: `I need to meet with you next week.`
- Stutter, false start, filled pauses:
  - Original input: `October the first I have a meeting on the uh I have a meeting at ten to eleven in the morning.`
  - Content input: `October first I have a meeting at ten to eleven in the morning.`

Ungrammaticality

One source of ungrammaticality is the limited coverage of the grammar. Any pre-written grammar is bound to fail in covering the wide range of spoken language used by different speakers, even when restricted to a specific domain. Utterances that are grammatical in the large sense may fail to be covered by the grammar in their entirety. The best one can hope for is for the grammar to cover a wide variety of expected grammatical constructs, and thus enable the parser to succeed in parsing the core part of the utterance, and ignore the rest as noise. Here are some examples, once again taken from recorded dialogs in the scheduling domain. Each example is an utterance that fails to be completely parsable by a recent version of our grammar. Along with each utterance, we present the most complete subset of the utterance that is parsable by the grammar.

- Unrecognized proper noun:
  - Original input: `I'm sorry Lynn.`
  - Parsable subset : `I'm sorry.`

- Unrecognized compound noun:
  - Original input: `I'm busy during those lunch hours from one to four on the twenty seventh.`
  - Parsable subset : `I'm busy during lunch from one to four on the twenty seventh.`
- Unrecognized complex construct:
  - Original input: `I just realized that I'm too busy on the first.`
  - Parsable subset : `I'm too busy on the first.`
- Unrecognized beginning:
  - Original input: `Listen, how about Thursday December second.`
  - Parsable subset : `How about Thursday December second.`
- Unrecognized verb form:
  - Original input: `I can't do lunch with you but we can meet later.`
  - Parsable subset : `I can't lunch with you but we can meet later.`

Another source of ungrammaticality arises due to the nature of the spoken language. People seem to adhere less to rules of the grammar when they speak (as opposed to when they write). Even so, the clauses and sentences that comprise the meaningful content of spoken utterances are for the most part structurally well formed, and can thus be considered "grammatical". The grammars that are developed for parsing spontaneous speech can describe the structure of these meaningful parts of the utterance. In cases where parts of the utterance deviate from what would be considered grammatical according to the grammar, the most we can expect of a parser is to extract and parse the parts of the utterance that are grammatical.

Dealing with Errors of the Speech Recognizer

Even though the quality of speech recognition systems has improved substantially over the last several years, a speech recognizer that handles spontaneous speech will frequently misrecognize words of the input utterance. Grammar based parsing is very sensitive to such misrecognitions. However, the speech recognition process produces multiple hypotheses for a given speech utterance. It is often the case that a highly accurate hypothesis exists, even though it is not the one considered best by the recognizer. Since a more accurate hypothesis is likely to be more grammatical and thus produce a better parse, we can hopefully use the parser to detect a hypothesis with fewer recognition errors. GLR* is suitable for this purpose, since it can parse hypotheses even if they are not completely grammatical. In Chapter 7 of the thesis, we investigate how this can be done efficiently using a lattice parsing version of GLR*.

1.2. Research Goals

This thesis was primarily motivated by the problem of parsing spontaneously spoken language. As illustrated in the examples of the previous section, spontaneous speech is often ill-formed. The spontaneous "on-line" nature of this type of input often results in disfluencies that for the most part can be regarded as noise that contaminates the meaningful segments of the utterance. Furthermore, the expression forms used by speakers are likely to exceed the language coverage of the grammar. However, we make the observation that the "core" parts of spoken utterances can be grammatically analyzed, and that in most cases, this analysis can reflect the essential meaning being conveyed. We therefore set out to demonstrate that a parser that can focus on the maximal grammatically analyzable subset(s) of a spoken input utterance, can substantially reduce the problems of noise contamination and limited grammar coverage, and that this results in a substantial improvement in the parser's ability to correctly analyze spontaneously spoken language.

We chose to base the developed parsing architecture on a standard efficient natural language parser, so that issues of robustness will be separated as much as possible from the development of the grammars, lexicons and other linguistic tools, and so that existing tools and grammars will be usable "as is". The GLR parsing system was chosen as an appropriate parsing architecture, since it is an efficient well developed architecture that has been in extensive use for large practical machine translation applications in our local research environment at CMU.

The major research goals set for the thesis were the following:

1. **To develop an efficient robust parsing algorithm**, that can find and parse the maximal grammatical subset of a given input utterance. Furthermore, **to couple the parsing algorithm with heuristics for search control**, that will allow it to perform within feasible time and space bounds.

2. **To develop a statistical disambiguation module**, that can effectively assist in selecting the correct analysis from the set of ambiguous analyses produced by the parser for a particular parsed input. An additional sub-goal was to develop a method that will allow the statistical scores assigned by the disambiguation module to be used in the evaluation of *different* parsable subsets of the input utterance, so that the "best" parse found by the parser can be selected.

3. **To develop a general framework for parse evaluation**, that can evaluate and rank sets of competing parse analyses that correspond to different grammatical subsets of the input, and which is capable of combining a variety of knowledge sources and heuristic evaluation measures. Additionally, **to develop a set of parse evaluation measures** that are suitable for evaluating the parses produced by the GLR* parser in the domain in which we have applied it - spontaneously spoken scheduling dialogs. Furthermore, in some cases, GLR* will have to skip input segments with crucial information. Thus, parsability alone is insufficient for ensuring that the analysis is meaningful. We therefore set out **to develop a parse quality heuristic**, that will allow the parser to self-judge the quality of the "best" analysis found, using the developed parse evaluation measures.

4. **To investigate the effectiveness and practical feasibility of the developed GLR* parser** in parsing spontaneous speech input, by integrating it into JANUS, a large speech-to-speech-

translation project, and by conducting extensive evaluations of the end-to-end performance of the system using GLR*.

5. **To construct a version of the robust GLR* parser that is capable of parsing lattices produced by the speech recognizer**, to integrate it into a lattice processing version of the JANUS system, and to investigate the ability of this architecture to overcome speech recognition errors and to improve the end-to-end performance of the system.

1.3. Thesis Summary

1.3.1. Foundation: GLR Parsing

The GLR* parsing algorithm developed in this thesis is based on Tomita's Generalized LR (GLR) parsing algorithm. The underlying GLR parsing algorithm and the foundations of the LR parsing method are described in Chapter 2 of the thesis.

Principles of LR Parsing

The GLR parsing algorithm is a parsing framework that evolved out of the LR parsing techniques that were originally developed for parsing programming languages in the late 1960s and early 1970s. LR parsers parse the input bottom-up, scanning it from left to right, and producing a rightmost derivation. They are deterministic and extremely efficient, being driven by a table of parsing actions that is pre-compiled from the grammar.

The common core of all of the LR parser variants is a basic Shift-Reduce parser, which is fundamentally no more than a finite-state pushdown automaton (PDA). The parser scans a given input left to right, word by word. At each step, the LR parser may either *shift* the next input word onto the stack, *reduce* the current stack according to a grammar rule, *accept* the input, or *reject* it. An action table that is pre-compiled from the grammar guides the LR parser when parsing an input. The action table specifies the next action that the parser must take, as a function of its current state and the next word of the input.

The GLR Parsing Algorithm

Tomita's Generalized LR parsing algorithm extended the original LR parsing algorithm to the case of non-LR languages, where the parsing tables contain entries with multiple parsing actions. The algorithm deterministically simulates the non-determinism introduced by the conflicting actions in the parsing table by efficiently pursuing in a pseudo-parallel fashion all possible actions. The primary tool for performing this simulation efficiently is the use of a *Graph Structured Stack* (GSS). Two additional techniques, *local ambiguity packing* and *shared parse forests*, are used in order to efficiently represent the various parse trees of ambiguous sentences when they are parsed.

The Unification-based GLR Parsing System

The Generalized LR Parser/Compiler is a unification based practical natural language system that was designed based on the GLR parsing algorithm at the Center for Machine Translation at Carnegie Mellon University. The system supports grammatical specification in an LFG framework,

that consists of context-free grammar rules augmented with feature bundles that are associated with the non-terminals of the rules. Feature structure computation is, for the most part, specified and implemented via unification operations. This allows the grammar to constrain the applicability of context-free rules. A reduction by a context-free rule succeeds only if the associated feature structure unification is successful as well. The Generalized LR Parser/Compiler is implemented in Common Lisp, and has been used as the analysis component of a wide range of projects at the Center for Machine Translation at CMU in the course of the last several years.

1.3.2. The GLR* Parsing Algorithm

Chapter 3 of the thesis describes the GLR* parsing algorithm, analyzes its computational complexity and evaluates its practical performance. It also describes the search control heuristics that were developed to ensure that the algorithm performs within feasible time and space bounds, and evaluates the effectiveness of these heuristics.

The GLR* parsing algorithm was designed to be robust to two particular types of extra-grammaticality: noise in the input, and limited grammar coverage. It attempts to overcome these forms of extra-grammaticality by ignoring the unparsable words and fragments and conducting a search for the maximal subset of the original input that is covered by the grammar.

The Unrestricted GLR* Parsing Algorithm

GLR* accommodates skipping words of the input string by allowing shift operations to be performed from inactive state nodes in the GSS. Shifting an input symbol from an inactive state is equivalent to skipping the words of the input that were encountered after the parser reached the inactive state and prior to the current word that is being shifted. Since the parser is LR(0), previous reduce operations remain valid even when words further along in the input are skipped, since the reductions do not depend on any lookahead.

Due to the word skipping behavior of the GLR* parser, local ambiguities occur on a much more frequent basis than before. In many cases, a portion of the input sentence may be reduced to a non-terminal symbol in many different ways, when considering different subsets of the input that may be skipped. Since the ultimate goal of the GLR* parser is to produce maximal (or close to maximal) parses, the process of local ambiguity packing can be used to discard partial parses that are not likely to lead to the desired maximal parse. Local ambiguities that differ in their coverage of an input segment can be compared, and when the word coverage of one strictly subsumes that of another, the subsumed ambiguity can be discarded.

Complexity Analysis and Performance Evaluation

The complexity analysis of the unrestricted GLR* algorithm proves that the asymptotic time complexity of the algorithm is $O(n^{p+1})$, where n is the length of the input, and p is the length of the longest grammar rule. This complexity bound is similar to that of the original GLR parsing algorithm. This result may seem surprising, but we provide several explanations for it. With respect to the grammar size, GLR has a non-polynomially-bound runtime, and once again, GLR* has a similar bound. These complexity results thus do not provide much insight into the actual differences in runtime behavior between GLR and GLR* on common practical grammars. We

therefore conducted experiments to evaluate the practical performance of the unrestricted GLR* algorithm.

Time and space performance of the unrestricted GLR* parser was evaluated on a benchmark setting from the JANUS speech-to-speech translation project. We used a version of the English analysis grammar for the scheduling domain. The benchmark consists of 552 English sentences, with sentence length ranging from 2 to 15 words, and an average length of about 6 words per sentence. The evaluation results show that the time and space behavior of the unrestricted GLR* parser diverges very rapidly from that of GLR, and becomes infeasible for even sentences of medium length. Whereas GLR time and space requirements increase almost linearly as a function of the sentence length, the performance of unrestricted GLR* appears to be similar to that predicted by the worst case complexity analysis. This suggests that effective search heuristics are required to ensure that the algorithm performs within feasible time and space bounds.

The Search Control Heuristics

Since the purpose of GLR* is to find only maximal (or close to maximal) parsable subsets of the input, we can drastically reduce the amount of search by limiting the amount of word skipping that is considered by the parser. We developed and experimented with two heuristic search techniques. With *the k-word Skip Limit Heuristic*, the parser is restricted to skip no more than k consecutive input words at any point in the parsing process. With the *Beam Search Heuristic*, the parser is restricted to pursue a "beam" of a fixed size k of parsing actions, which are selected according to a criterion that locally minimizes the number of words skipped. There exists a direct tradeoff between the amount of search (and word skipping) that the parser is allowed to pursue and the time and space performance of the parser itself. The goal is to determine the smallest possible setting of the control parameters that allows the parser to find the desired parses in an overwhelming majority of cases.

We conducted empirical evaluations of parser behavior with different settings of the search control parameters, using the same benchmark setting on which the unrestricted GLR* parser was evaluated. The results show that time and space requirements of GLR* increase as the control parameters are set to higher values. This increase however is gradual, and in the case of the beam parameter, even with considerably high settings, parser performance remains in the feasible range, and does not approach the time and space requirements experienced when running the unrestricted version of GLR*. A comparison of the performance figures of the two heuristic control mechanisms revealed a clear advantage of using the beam search, versus the simpler skip word limit heuristic.

1.3.3. Statistical Disambiguation

The abundance of parse ambiguities for a given input is greatly exacerbated in the case of the GLR* parser, due to the fact that the parser produces analyses that correspond to many different maximal (or close to maximal) parsable subsets of the original input. It is therefore imperative that the the parser be augmented with a statistical disambiguation module, which can assist the parser in selecting among the ambiguities of a particular parsable input subset. Chapter 4 describes a newly developed statistical disambiguation module, designed to augment the unification-based versions of both the GLR and GLR* parsing systems.

The Statistical Framework

Our probabilistic model is based on a similar model developed by Carroll, where probabilities are associated *directly* with the actions of the parser, as they are defined in the pre-compiled parsing table. Because the state of the LR parser partially reflects the left and right context of the parse being constructed, modeling the probabilities at this level has the potential of capturing contextual preferences that cannot be captured by probabilistic context-free grammars. Because we use a finite-state probabilistic model, the action probabilities do not depend on the given input sentence. The relative probabilities of the input words at various states is implicitly represented by the action probabilities.

Training the Probabilities

The correctness of a parse result in our parsing system is determined according to the feature structure associated with the parse and not the parse tree itself. We are therefore interested in parse disambiguation at the feature structure level. To allow adequate training of the statistical model, we collect a corpus of sentences and their correct feature structures. Within the JANUS project, a large collection of sentences and their correct feature structures had been already constructed for development and evaluation purposes, and was thus directly available for training the probabilities of the statistical model. For other domains (including ATIS), we developed an interactive disambiguation procedure for efficiently constructing a corpus of disambiguated feature structures.

Once we have a corpus of input sentences and their disambiguated feature structures, we can train the finite-state probabilistic model that corresponds to the LR parsing table. In order for this to be done, a method was developed by which a correct feature structure of a parsed sentence can be converted back into the sequence of transitions the parser took in the process of creating the correct parse. Using this method, we process the training corpus and accumulate counters that keep track of how many times each particular transition of the LR finite-state machine was taken in the course of correctly parsing all the sentences in the training corpus. After processing the entire training corpus and obtaining the counter totals, the counter values are treated as action frequencies and converted into probabilities by a process of normalization. A smoothing technique is applied to compensate for transitions that did not occur in the training data.

Runtime Parse Disambiguation

Statistical disambiguation for the unification-based GLR parsing system is performed as a post-process, after the entire parse forest has been constructed by the parser. At runtime, the statistical model is used to score the alternative parse analyses that were produced by the parser. Once computed, these scores can be used for disambiguation, by unpacking the analysis that received the best probabilistic score from the parse forest.

Due to ambiguity packing, the set of alternative analyses and their corresponding parse trees are packed in the parse forest. For packed nodes, a statistical score is computed for each of the alternative sub-analyses that are packed into the node. The packed node itself is then assigned the "best" of these scores, by taking a maximum. This score will then be used to compute the scores of any parent nodes in the forest. According to this scheme, the score that is attached to the root node of the forest should reflect the score of the best analysis that is packed in it. Because the score computation of an internal node in the parse forest requires the prior computation of the scores of

the node's children, parse node scoring is performed in a "bottom-up" fashion. Because of some effects of packing and unification, the score assigned to a parse node is not guaranteed to be the actual true probability of the best sub-analysis rooted at the node. However, the score is always a close over-estimate of this probability. It may thus be viewed as a heuristic for determining the actual best scoring ambiguity packed in the forest.

Once the preliminary task of scoring all the nodes in the parse forest is complete, the actual parse disambiguation can be performed by a process of unpacking the best scoring analysis from the forest. While the node scoring was done in a "bottom-up" fashion, unpacking is done "top-down", starting from the node at the root of the forest. Unpacking is done by going down the parse tree, and selecting one of the alternative packed ambiguities at each packed parse node.

Evaluation

We evaluated the statistical disambiguation module for two large practical grammars. The first is a syntactic grammar developed for the ATIS domain. The second grammar is the Spanish analysis grammar developed for the JANUS/Enthusiast project. It is a predominantly semantic grammar (with some syntactic structural rules) for the appointment scheduling domain.

To train the probabilities for the ATIS grammar, we used the interactive training procedure, where for each sentence, the user is required to select the correct analysis (feature structure) from the set of analyses produced by the parser. Because this task is time consuming, training was conducted on a set of only 500 sentences, from which a smoothed probability table was constructed. We then tested the disambiguation module on an unseen set of 119 sentences. 67 out of the 119 test sentences (56.3%) are ambiguous. In 52 out of these 67 ambiguous sentences (77.6%), one of the analyses produced by the parser is the correct analysis. The disambiguation module selected the correct analysis in 39 out of the 52 sentences, resulting in a success rate of 75.0%. Without statistical disambiguation, the parser is capable of selecting the correct parse in 30 of the 52 sentences (57.7%), by simply choosing the first analysis out of the set of analyses returned by the parse. This is a good result, when we consider the fact that the size of the training set was inadequately small (the training observed a mere 4.4% of the possible actions, and no actions at all were observed for 60% of the states).

For the Spanish analysis grammar, the probabilities for statistical disambiguation were trained on a corpus of "target" (correct) ILTs of 1687 sentences. We then evaluated the performance of the statistical disambiguation module on a unseen test set of 769 sentences. 524 out of the 769 sentences (68.1%) are parsable with one of the analyses returned by the parser being completely correct. 213 of these 524 sentences (40.6%) are ambiguous. The statistical disambiguation module successfully chose the correct parse in 140 out of the 213 sentences, thus resulting in a success rate of 65.7%. For comparison, we applied a Minimal Attachment disambiguation procedure to the same test set. With Minimal Attachment, the correct parse was chosen in only 79 out of the 213 sentences, resulting in a success rate of 37.1%. Thus, the statistical disambiguation method outperforms the Minimal Attachment method in this case by about 29%. Although the size of the corpus that was available for training for the Spanish grammar was about three times larger than that used for the ATIS grammar, we still observed and modeled only a small fraction of the actual transitions allowed by the grammar. The training observed only 5.3% of all possible actions, and for 59% of the states, no actions were observed.

The statistical disambiguation results for both grammars demonstrate that even extremely small amounts of correct training data can establish structural preferences for resolving the most frequently occurring types of ambiguity. Thus, significant benefits can be gained from using the statistical disambiguation module even when trained on relatively small amounts of data.

1.3.4. Parse Evaluation Heuristics

Chapter 5 describes an integrated framework that we have developed for the GLR* parser, that allows different heuristic parse evaluation measures to be combined into a single evaluation function, by which sets of parse results can be scored, compared and ranked. The framework itself is designed to be general, and independent of any particular grammar or domain to which it is being applied. The framework allows different evaluation measures to be used for different tasks and domains. Additional evaluation measures can be developed and added to the system. The evaluation framework includes tools for optimizing the performance of the integrated evaluation function on a given training corpus.

The utility of a parser such as GLR* obviously depends on the semantic coherency of the parse results that it returns. Since the parser is designed to succeed in parsing almost any input, parsing success by itself can no longer provide a likely guarantee of such coherency. We thus developed a classification heuristic by which the parser tries to self-identify situations in which even the "best" parse result is inadequate.

The Integrated Framework for Parse Evaluation

The integrated parse evaluation framework is a method for combining different evaluation measures into a single evaluation function. Each of the available evaluation measures is called a score function. A score function can reflect any property of either a parse candidate or the skipped (or substituted) portions of the input that correspond to a parse candidate. We use linear combination as the scheme for combining the individual score functions. With linear combination, each of the individual scores are first scaled by a weight factor, and the results are then summed.

Selecting the weight assigned to each of the score functions allows us to intentionally enforce the dominance of certain evaluation measures over others. To fine tune these weights, we developed a semi-automatic training procedure, which attempts to find the optimal combination weights for a given training corpus. The goal of the optimization procedure is to adjust the weights of the score functions in a way that maximizes the number of sentences in the training corpus for which the correct parse is assigned the best score. This is done by analyzing the set of sentences for which the correct parse was not assigned the best score. These sentences are classified into error vectors, which reflect which weights should be increased and which should be decreased. The optimization procedure works iteratively. It is only semi-automatic, due to the fact that it consults with the user before actually modifying the weights of the score functions, and proceeding to the next iteration.

The Parse Evaluation Score Functions

For both the JANUS scheduling domain and the ATIS grammar, we developed a set of four evaluation measures: a penalty function for skipped words, a penalty function for substituted

words, a penalty function for the fragmentation of the parse analysis, and a penalty function based on the statistical score of the parse.

In its basic form, the penalty for skipping words assigns a penalty in the range of $[0.95, 1.05]$ for each skipped word. This scheme assigns a slightly higher penalty to skipped words that appear later in the input. This was designed to help us to deal with false starts and repeated words. This simple scheme was further developed into one in which the penalty for skipping a word is a function of the saliency of the word in the domain. To determine the saliency of our vocabulary words, we compared their frequency of occurrence in transcribed *domain* text with the corresponding frequency in "general" text.

For the ATIS domain, we attempted to deal with substitution errors of the speech recognizer using a confusion table. Words that are common misrecognitions of other words are listed in the table along with one or more words that are likely to be their corresponding correct word. When working with a confusion table, whenever the GLR* parser encounters a possible substitution that appears in the table, it attempts to continue with both the original input word and the possible "correct" word(s). A word substitution should incur some penalty, since we would like to choose an analysis that contains a substitution only in cases where the substitution resulted in a better parse. We currently use a simple penalty scheme, where each substitution is given a penalty of 1.0.

In terms of their grammatical restrictness, our grammars for parsing spontaneous speech are rather "loose". The major negative consequence of this looseness is a significant increase in the degree of ambiguity of the grammar. In particular, utterances that can be analyzed as a single grammatical sentence, can often also be analyzed in various ways as collections of clauses and fragments. Our experiments have indicated that, in most such cases, a less fragmented analysis is more desirable. We thus developed a method for associating a numerical fragmentation value with each parse. This fragmentation value is then used as a penalty score.

Our statistical disambiguation uses a finite-state model that assigns an actual probability to every possible transition sequence. Thus, transition sequences that correspond to parses of different inputs are directly comparable. This enables us to use the statistical score as a parse evaluation measure. To enable parse probabilities to be used as an evaluation measure, we convert the probability into a penalty score, such that more "common" parse trees receive a lower penalty.

The Parse Quality Heuristic

We developed a classification heuristic that is designed to try and identify situations in which even the "best" parse result is inadequate. Our parse quality heuristic is designed to classify the parse chosen as best by the parser into one of two categories: "Good" or "Bad". The classification is done by a judgement on the combined penalty score of the parse that was chosen as best. The parse is classified as "Good" if the value of penalty score does not accede a threshold value. The value of the threshold is relative to the length of the input utterance, since longer input utterances can be expected to accommodate more word skipping, while still producing a good analysis.

Performance Evaluation

We evaluated the performance of the parse selection heuristics and the parse quality heuristic in two different settings. The first setting was an unseen test set of 513 transcribed Spanish dialogs in the scheduling domain, using the Spanish analysis grammar developed for the JANUS/Enthusiast

project. The second setting is a unseen test set of 120 sentences in the ATIS domain, using the ATIS syntactic grammar.

In the Spanish grammar evaluation, 500 out of the 513 sentences are parsable, and for 300 out of the 500 parsable sentences, one of the parses produced by the parser completely matches its target ILT. In 33 cases, the correct parse requires some word skipping. With the full set of parse evaluation heuristics, 26 of the 33 cases (78.8%) were correct. A simple evaluation heuristic that selects the maximal parsable subset (breaking ties randomly) finds the correct parse in 24 of the 33 cases (72.7%). 200 out of the 500 parsable test sentences do not produce a completely correct ILT. 148 out of the 200 are sentences that require some amount of skipping in order to be parsed. With the simple parse selection heuristic, the parser selects a parse that results in an acceptable translation in 72 out of the 148 sentences. With the full set of heuristics, an acceptable translation is produced in 78 out of the 148 cases. Additionally, we found that in only 6 additional cases there exists an analysis that would produce an acceptable translation that was not selected by either of the methods.

We also evaluated the parse quality heuristic for the Spanish grammar. 57 out of the 500 parsable sentences are marked as "Bad" by the heuristic. None of the 57 sentences marked "Bad" have a completely correct parse that matches the corresponding target ILT. For 47 out of the 57 sentences marked "Bad" (82.4%), the selected parse produces a bad translation. Of the 443 sentences marked "Good" by the parse quality heuristic, the selected parse for 274 of the sentences (61.9%) matches its target ILT, and for 115 of the sentences (26.0%), the selected parse is not a complete match, but produces an acceptable English translation. Thus, the translation accuracy success rate for parses marked as "Good" by the parse quality heuristic is 87.9%.

The results for the ATIS evaluation were rather similar. 119 of the 120 test sentences were parsable. The simple parse selection heuristic selects an acceptable parse for 69 out of the 119 parsable sentences (58.0%). The full parse selection heuristics select an acceptable parse in 4 additional cases, for a total of 73 out of 119 (61.3%). 23 out of the 119 parsed sentences are marked as "Bad". For 21 of these 23 sentences (91.3%), the selected parse is indeed bad. Also, in only 14 cases (14.6%) did the parse quality heuristic fail to mark a truely bad parse as "Bad".

1.3.5. Parsing Spontaneous Speech using GLR*

Chapter 6 describes how the GLR* parser has been integrated into JANUS, a practical speech-to-speech translation system, designed to facilitate communication between a pair of different language speakers, attempting to schedule a meeting. We also evaluate the performance of GLR* on speech recognized input in the ATIS domain.

Parsing Full Utterances with GLR*

In JANUS, the output of the speech recognizer is a textual hypothesis of the complete utterance, and contains no explicit representation of the boundaries between the clauses and sentences of which the utterance is composed. The analysis grammars developed for the scheduling domain were constructed to primarily analyze clauses. Multi-clause utterances must therefore be broken into clauses, in order to be properly parsed. We handle multi-clause utterances using a combination of methods. First, the analysis grammars were extended with rules that allow the input utterance to be analyzed as a concatenation of several clauses. However, for typical multi-clause utterances,

there are many different possible ways of breaking the utterance into clauses that are analyzable by the grammar. For long multi-clause utterances, the computational complexity of pursuing all such possibilities results in infeasible parse times and memory requirements. We significantly reduce the severity of this problem by using a pre-processor to break the utterance into smaller sub-utterances, that may still contain a number of clauses. In addition, we developed a powerful set of heuristics, including a statistical clause boundary predictor, for further constraining the breaking possibilities that are considered by the parser.

JANUS Performance Evaluation

Evaluations were conducted on both the English and Spanish analysis grammars. We compared the performance of the system incorporating GLR* with a version that uses the original GLR parser, and with a version that uses the Phoenix parser. The results for both the Spanish and English are rather similar, and we thus summarize here only the English results.

The test set consisted of 99 push-to-talk unseen utterances. The transcribed text was pre-broken into clauses according to transcribed markings. This produced a set of 360 sub-utterances or clauses, each of which was parsed separately. The top-best speech recognized hypotheses were pre-broken using the two consecutive noise words criterion. This produced a set of 192 sub-utterances.

On transcribed input, GLR succeeds to parse 55.8% of the clauses, while GLR* succeeds in parsing 95.0% of the clauses. This amounts to a gain of about 39% in the number of parsable clauses. On the speech recognized input, GLR parses only 21.4% of the input sub-utterances, while GLR* succeeds to parse 93.2%. In this case, the increase in parsable sub-utterances amounts to almost 72%. On transcribed input, GLR* produced acceptable translations for 85.6% of the clauses, compared with 54.2% for GLR. Thus, GLR* produces over 30% more acceptable translations. This is also the case when only the parses marked "Good" by the parse quality heuristic of GLR* are considered. The results on speech recognized input are rather similar. While GLR produced an acceptable translation only 17.2% of the time, GLR* did so in 47.9% of the time. Once again, this amounted to an increase of about 30.0% in the number of acceptable translations. Similar results were achieved when only the parses marked "Good" by GLR* are considered.

We also evaluated the effectiveness of the parse quality heuristic. On transcribed data, 92.7% of the parses marked "Good" by the parse quality heuristic produce acceptable translations. 10 out of 12 clauses marked by GLR* as "bad" (83.3%) result in a bad translation. On speech data, the effectiveness of the parse quality heuristic is hindered by errors of the speech recognizer. Consequently, parses marked as "Good" by the parse quality heuristic, produced acceptable translations for only 66.9% of the sub-utterances. However, when errors that are completely due to poor recognition are considered acceptable, 96.2% of the clauses marked "Good" by the parse quality heuristic are in fact acceptable. Parses marked "Bad" by the parse quality heuristic produced a bad translation in 93.5% of the time.

The performance of GLR* and Phoenix on this test set was quite similar. On both transcribed and speech input, the percentage of utterances with acceptable translations using Phoenix was slightly better than the corresponding figure when using GLR*. On transcribed input, GLR* produces acceptable translations for 83.9% of the utterances, versus 85.5% for Phoenix. On speech input, GLR* has acceptable translations for 44.5% of the utterances, while Phoenix has acceptable translations for 45.8%.

GLR* Performance on ATIS Domain

For the ATIS evaluation, we used a set of 120 unseen speech recognized sentences. The portion of parsable sentences increased from 51.7% for GLR to 99.2% for GLR*. However, the portion of parsable sentences with an acceptable parse increased only by 10.8% (from 50.0% for GLR, to 60.8% for GLR*). The evaluation of the parse quality heuristic shows that while the portion of bad parses out of the parses marked "Bad" by the parser is still very high (91.3%), the parse quality heuristic is somewhat less effective on parses marked "Good". Only 74% of the parses marked "Good" were in fact acceptable. Further inspection, however, indicated that 11 marked "Good" sentences had bad parses due to incorrect ambiguity resolution. If the correct ambiguity were chosen for these parses, the portion of good parses out of parses marked "Good" would reach 85%, similar to the comparable figures found in the JANUS evaluations.

1.3.6. Parsing Speech Lattices using GLR*

In practice, the top-best hypotheses produced by our speech recognition system are far from perfect. Speech recognition errors hinder on the ability of the parser to find a correct analysis for the utterance, and the effects of this toll are reflected in the disparity between our performance results on transcribed and speech recognized input. Processing multiple speech hypotheses instead of a single hypothesis thus has the potential of detecting a hypothesis with fewer recognition errors, which should lead to an improvement in the overall translation performance. In Chapter 7, we investigate how GLR* can be adapted into a parser capable of directly parsing speech produced word lattices.

The Advantages of Parsing Speech Lattices

Parsing the speech lattice directly attempts to efficiently accomplish the same results of parsing an n-best list. Instead of first extracting the list of hypotheses from the speech lattice, and then parsing each of them separately, we use a version of the parser that can parse the lattice directly. Each word in the lattice is parsed only once, although it may contribute to many different hypotheses. The lattice parser produces a large set of possible parses, corresponding to the parses of various complete word paths through the lattice. This set of parses can be scored and ranked according to the parse score, or according to some optimized combination of the parse score and recognizer score.

The GLR* Lattice Parsing Algorithm

For the lattice parser to function correctly, it is essential that the words of the lattice be parsed in a proper order. If a lattice word $[w_1(i_1, j_1)]$ appears prior to $[w_2(i_2, j_2)]$ in some path through the lattice, then they must be parsed in that order. A simple procedure can verify that the lattice words are properly ordered prior to parsing.

In the lattice parsing version of GLR*, the shift distribution step was modified to handle lattice input words. Similar to the string parsing version of GLR*, shift actions must first be distributed to state nodes that do not result in any new skipped words. Subsequently, if the number of shift actions distributed has not already acceded the beam-width, other inactive state nodes are considered as well. State nodes that result in fewer skipped words are considered first. This is done by first

detecting all levels that correspond to words of distance one from the current word. Shift actions are then distributed to state nodes of these levels. If the total number of shift actions distributed is still below the beam-width, we proceed to find the levels that correspond to words of distance two, and distribute shift actions to the state nodes of these levels as well. This process continues until the total number of distributed shift actions reaches the beam-width.

When dealing with a general word lattice, determining word connectivity and lattice paths is more complicated. For two arbitrary words $[w_1\ (i_1, j_1)]$ and $[w_2\ (i_2, j_2)]$ in the lattice, there could be zero, one, or multiple paths of lattice words that connect them. We thus developed a procedure for determining the connectivity of two points in the lattice.

An additional field was added to the symbol node structure, in order to keep track of the set of lattice words that are "covered" by the symbol. This set contains all the words that were actually parsed in the sub-parse tree rooted by the symbol. The ambiguity packing procedure of the parser was also slightly modified. Two symbol nodes of the same grammar category can be packed by the lattice parser only if they both correspond to a parse of the exact same sub-path in the lattice.

The Lattice Processing Version of JANUS

The lattice parsing version of GLR* has been incorporated into a lattice processing version of the JANUS system. This required some changes in the configuration of the system. For each input utterance, instead of producing a top-best hypothesis or an n-best list, the speech recognition module outputs a complete scored word lattice. The lattices produced by the speech recognizer are usually far too large and redundant to be parsed in feasible time and space, and require significant pruning in order to be parsed. We use a lattice pre-processor for this task. The pre-processor performs four major functions. First, it collapses noise-words, and removes redundant paths that differ only in noise-words. Next, it breaks the lattice into sub-lattices according to pauses in the speech signal, which are highly correlated with clause boundaries. Third, the transformed sub-lattices are rescored with a language model. Finally, each sub-lattice is pruned to a parsable size using a method that ensures that the best scoring hypotheses are not pruned out. The resulting sub-lattices are then separately passed on to the lattice parser for parsing. A post-parsing procedure combines the results of parsing the sub-lattices into a ranked list of output results which is passed on to the discourse processor and to the generation module.

Performance Evaluation

We have recently conducted an evaluation to measure the performance of the lattice parsing approach. On a common test set of 14 unseen English dialogs (99 utterances), we compared the results of using the lattice processing version of the JANUS system with the results of processing the top-best speech hypothesis. The preliminary results show only a slight improvement in the total percentage of acceptable translations, which increased from 41.4% for parsing the top-best speech hypothesis, to 44.5% when parsing lattices. This compares with a performance of 83.9% acceptable translations on the transcribed input of the same data. The lattice parser succeeded to produce an acceptable translation for 15 utterances that were badly translated due to poor recognition when the top-best speech hypothesis was used. However, 12 utterances that were acceptably translated by using the top-best speech hypothesis, were translated poorly when using the lattice parser. In most of these cases, this was due to the fact that the lattice parser prefered a hypothesis in the lattice

that produced a better parse score than the top-best speech hypothesis, even though it contained more speech recognition errors. Such problems could possibly be resolved by a better combination scheme between the acoustic score and the parser score, and by improvements in the parse score heuristics, and require further investigation.

1.4. Thesis Contributions

The major contributions of this thesis are the following:

1. The development of an efficient robust general parsing architecture, capable of significantly reducing the problems of noise contaminated input and limited grammar coverage.

2. The development of a general framework for evaluating and ranking sets of competing parse analyses, that is capable of combining a variety of knowledge sources and heuristic evaluation measures. Additionally, the development of a parse quality heuristic, that allows the parser to self-judge the quality of its output results.

3. The demonstration of the effectiveness and practical feasibility of the methods employed by GLR* in parsing spontaneous speech input, by substantially enhancing the end-to-end performance of the JANUS speech-to-speech translation system.

1.5. Previous Related Work

1.5.1. Other Approaches to Robust Parsing

There have been several different approaches to robust parsing in the last decade. To avoid failure when faced with extra-grammatical input, most approaches relax syntax and grammar constraints. In some cases, semantic information is used to compensate for the lack of grammaticality in order to partially or completely drive the analysis process.

Hayes and Mouradian [32] were among the first to try to tackle these kind of problems. They developed a pattern matching parser that parsed the input in a bottom-up fashion. Robustness is achieved by using a number of flexible pattern matching schemes and by allowing parses to be suspended when faced with unexpected input, and then reactivated with later input words. To limit the search for the best analysis as much as possible, the system uses an ordered set of parsing heuristics of increasing flexibility. For each parsed word, the system attempts to expand its existing partial parses using the more restrictive methods first. A more flexible method is applied only when the previous method failed to account for the new word.

In another early work, Kwasny and Sondheimer [50] attempt to handle input that contains segments that are unparsable by using a bottom-up ATN parser that can recognize grammatical fragments. The ATN grammar allows fragments to be accepted as complete analyses, but only at times where they are predicted by contextual information.

Lang [53] develops a parsing algorithm that is capable of parsing incomplete sentences that contain segments of one or more unknown words, and can efficiently represent the parse forest for such sentences. Although Lang suggests that the principles of this method should prove useful in the analysis of ill-formed input, it is not clear how this can be done in a practical way.

Other systems such as BBN's DELPHI system [89] and the MIT ATIS system [85] construct fall-back components that are designed to handle extra-grammatical input that causes the main parser to fail. The DELPHI system incorporates both syntax and frame-based fragment combination components. In the MIT system, grammatical constraints are relaxed when an input fails to parse in full, and the system attempts to combine partial parsed fragments, in search of the parse that consumes the most words of the original sentence. Weischedel and Black [105] describe a method for relaxing predicates in an ATN grammar which check for noun-verb agreement and other syntactic or semantic constraints.

Jensen and Heidorn's PLNLP system, and it's comprehensive syntactic English grammar PEG [38] use a "fitting" procedure to produce a reasonable parse structure for sentences that cannot be completely grammatically parsed. The fitting heuristic attempts to combine the largest parsed constituents into a complete structure by first selecting a head constituent and then attaching the remaining constituents to the head using some heuristic rules.

A different method for completing partial parses is suggested by Nasukawa [73]. Information from well-formed analyzed sentences in the surrounding text is used to reanalyze the part-of-speech and modifiee-modifier relationships of each word in the ill-formed sentence. If the resulting partial parses can still not be combined into a grammatical structure, the "discourse" information is once again used in order to combine the partial parses in a way that is most consistent with similar structures in the surrounding text. An implementation of the method in a machine translation system improved the accuracy of the translations produced by the system.

Another notable attempt to handle ill-formed input by a general purely syntactic method is presented by Mellish [68]. Mellish describes how to augment a chart parser with a procedure that can focus on an error in the input once the parser has failed. The error is detected by performing a top-down search through the chart left behind by the parser. Although the method can potentially handle missing words as well as misspelled and extra words, it is demonstrated to work reasonably well only in the case of a single error, and is not expected to generalize well to multiple errors.

Strzalkowski [90] describes TTP, a syntactic top-down parser implemented in Prolog, that is extremely fast, and uses a time-out mechanism that can skip unrecognized segments of the input and segments that were not parsable within preset time bounds. A stochastic tagger is used in an attempt to resolve lexical ambiguities prior to parsing, by assigning part-of-speech tags to the input words. The skipping and time-out parameters must be encoded explicitly into the rules of the grammar, and the parser's behavior and output quality are very sensitive to these encodings. This appears to considerably complicate the task of grammar development for the parser, and hinders on the generality of the TTP approach.

Carbonell and Hayes [10], in an early work that attempts to deal with a wide range of extra-grammatical phenomena, suggest that a semantic case-frame approach to parsing is the suitable way to handle many types of extra-grammaticality. Semantic interpretation of the input is achieved by detecting the main semantic concept of the sentence, and then searching the sentence for components that instantiate the semantic frames that are associated with the main concept. Hayes et. al. [31] show how these ideas can be specifically applied to parsing spoken language. This approach is very flexible to issues such as the order of appearance of the semantic frames in the input. However, it is domain dependent and has difficulties in capturing syntactic and other types of grammatical knowledge that is contained in the input.

McDonald [66] [65] describes an approach based on chart parsing, that is tailored for extracting domain specific information from unrestricted text. The highlight of the system is in its hybrid

approach, which combines several different analysis tools. It uses a succession of six different algorithms: a tokenizer, a finite-state acceptor, a context-free phrase structure parser, a context-sensitive parser, a conceptual analyzer and a phrase boundary prediction heuristic. All components access a common chart data structure which records their partial results and facilitates communication. The context-free and context-sensitive parsers use semantic grammars to combine the lower level phrases into phrases that represent semantic concepts. These are then combined into a coherent analysis by the conceptual analyzer, which allows gaps of unanalyzed segments of text between the combined phrases. Similar ideas can be found in Jacobs and Rau's SCISOR information extraction system [35]. The system combines bottom-up syntactic parsing with top-down conceptual expectation-driven parsing into a flexible multi-layer parser. One drawback of these systems is their complexity. It is not clear that robust parsing of spoken language in fact requires a system of such complexity. Consequently, one of the goals of this thesis is to investigate whether reasonable robustness can be achieved using a single parser based on the simpler architecture of the GLR parsing algorithm.

Menzel [69] suggests a unified approach for achieving both syntactic and semantic robustness in language analysis. He proposes using the Constraint Grammar formalism [41] to express syntactic, semantic and pragmatic linguistic constraints. The violation of constraints incurs penalties, and the importance of satisfying a constraint, as well as the relative importance of various constraints can be modeled via penalty weights. The system can then search for an analysis (or interpretation) that is minimal in overall penalty, and thus satisfies the constraints in the best possible way. Although Menzel supports this approach with several examples, the approach has apparently not been fully implemented or applied to a large scale application. However, the principle of using a weighted penalty scheme to select the "best" analysis is central to our GLR* approach as well.

1.5.2. Other Work on Parsing Speech

The dramatic improvement in speech recognition technology in the last decade has attracted a significant amount of research effort to spoken language processing and understanding. A variety of speech processing systems have addressed the issues of parsing spoken language in general, and the output of speech recognition systems in particular. Many speech processing systems have attempted to use standard text parsing algorithms (such as GLR and chart parsers), while others have applied some of the more robust parsing techniques described in the previous subsection. Saito and Tomita [83], for example, describe how the standard GLR parser can be extended to parse noisy input produced by a speech recognizer when parsing Japanese on a phoneme level. Insertions, deletions and substitutions of phonemes are considered by the parser, and are restricted by the set of phonemes predicted by the parsing table at each stage.

The interface between the speech recognition component and language processing is also a major issue in spoken language systems. While some systems use a simple sequential approach in which the top-best hypothesis of the speech recognizer (or a ranked n-best list) is passed on to a parser, other systems have attempted a closer integration of the speech and language processing components, by applying parsing directly to the word lattice produced by the speech recognizer. An overview of the most common approaches to this problem can be found in [29] and [102]. The GLR* parser described in this thesis is applicable to both of these configurations, and the issues involved in the integration of the speech and parsing components of such speech processing systems is addressed in greater depth in Chapter 7.

We next summarize three representative recent speech parsing systems with different approaches, and then mention some other parsing approaches, specifically designed to parse speech lattices.

The Spoken Language Translator

One system that follows the sequential integration approach is the Spoken Language Translator (SLT) being developed by SRI, SICS and Telia Research [1]. The speech recognition component produces a set of hypotheses that is then passed on to the SRI Core Language Engine, a general natural language processor. Each hypothesis is analyzed into a set of possible logical-form representations, and the entire set of possible analyses is ranked by a parse evaluation component, where the numerical scores attached to each analysis reflect it's linguistic plausibility. The most unique aspect of the language processor is an explicit repair module, which attempts to identify typical speech disfluencies (such as false starts and repetitions) in the input hypothesis and "correct" them. Input that cannot be parsed in its entirety is handled by a constraint relaxation method. The system also includes a corpus-based grammar specialization module which uses a training corpus to create a restricted version of the original grammar that is tailored to specifically cover only structures that appeared in the training corpus. The authors claim that using the restricted version of the grammar decreased the parsing time by two orders of magnitude, at the expense of only a 7% loss in coverage.

Hipp's Minimum-distance Parser

In his PhD thesis [33], Hipp describes a parser specifically designed for a spoken language dialog system. A minimum-distance parsing algorithm is used to analyze the input utterance, and the analysis is then converted into an internal representation using syntax-directed translation. When the input utterance is not grammatical, the parser searches for the analysis of a grammatical string which is "closest" to the original utterance. The method in principle allows deletions, substitutions and insertions to be considered, with the "closest" distance being determined by using adjustable penalties. The parsing algorithm is a bottom-up chart-based procedure, in which the set of "closest distance" parses is incrementally constructed from "closest distance" sub-parses. The parsing algorithm is designed to work with both string input and speech lattices. Although implemented, the parser was tested only within a limited small scale application, operating on top-best output of a 125 word vocabulary speech recognition system. No details are provided with respect to the grammar used in the experiments, or the actual time and space performance of the parser. Hipp acknowledges that it is an open question whether or not his approach can scale up feasibly to a much larger application. Thus, even though Hipp's approach is similar in goal to that of GLR*, and the two systems are even similar in some of their features, it was not possible to thoroughly compare their performance.

The Phoenix Parser and MINDS System

The Phoenix parser [103] [63] uses the semantic case-frame approach in order to achieve robustness in the analysis of spoken language. Semantic relations are represented by concept frames. Slots in a frame represent the semantically relevant information and are filled by matching patterns in an input string. The patterns which fill slots are represented as Recursive Transition Networks (RTNs).

The parser matches as much of the input utterance as possible to the patterns specified by the RTNs. The parser selects the analysis that is highest in coverage and least in fragmentation. The Phoenix approach has been proven to perform very well in tasks such as ATIS [104]. In several applications, the Phoenix parser has been closely coupled with the MINDS system [104] [111], a system that can dynamically apply linguistic and pragmatic constraints from the discourse to improve spoken language understanding performance. A version of the MINDS system designed for the resource management domain [111] uses dialog knowledge and a model of the user's goals, plans and focus to generate a set of predictions. These are then used to constrain both the speech search as well as the grammar used by the Phoenix parser. In the context of a system for the ATIS task [104], the constraints are used to detect and correct erroneous parses, skipped or overlooked information, and out of domain requests. The Phoenix parser is also used as an alternative parsing architecture to GLR* within the JANUS speech-to-speech project at CMU. Performance evaluations that include a detailed comparison of Phoenix and GLR* are presented in Chapter 6 of this thesis.

Parsing Speech Lattices

Several works have attempted to apply standard parsing algorithms to speech lattices. Chow and Roukos [12] show how to apply a chart parser to a speech lattice, using a dynamic programming procedure to associate the best score with each grammar phrase found. Tomita [93] describes how the GLR parsing algorithm can be modified for parsing lattices, so that only parses of completely grammatical hypotheses in the lattice are produced. Staab [88] describes how the efficiency of this approach can be increased by augmenting it with a beam search method. Nakagawa [72] uses a time-synchronous top-down parser to achieve a similar goal. Saito [82] describes how a speech lattice can be parsed efficiently in an island-driven fashion starting from "anchor" words (words with highly confident speech scores) using a bi-directional version of the GLR parser. Lavie and Tomita [56] describe a word-order-free version of the GLR parsing algorithm and show how it can be used in conjunction with an A* heuristic to efficiently find and parse the best scoring grammatical speech hypothesis in the speech lattice.

Not much research has been done into applying some of the more robust parsing methods to lattice parsing. Hanrieder [28] parses the lattice using a chart-based island parser, which assigns a quality score to each generated sub-parse. The scoring function integrates acoustic, syntactic and semantic quality measures. The parser then searches for the best scoring hypothesis, which can be a collection of partial parses. Giachin and Rullent [78] describe a system that parses a speech lattice using pre-compiled knowledge sources that combine both syntactic and semantic constraints. In Chapter 7 of this thesis, we investigate how a lattice parsing version of GLR* can improve the translation quality of the JANUS speech-to-speech translation system.

Several other works have attempted to integrate grammatical constraints directly into the speech recognizer's search and scoring. Hauenstein and Weber [30] investigate several methods for applying the constraints of a unification based chart parser directly into the speech search. Similar approaches are investigated by Ney [74] and Okada [77]. Murveit and Moore [71] [70] describe how a dynamic grammar network can restrict an HMM based speech recognizer to consider only grammatical hypotheses. Kita [46] uses the LR parsing table of a GLR parser to predict words, which are then used to constrain an HMM based speech recognizer. The MIT Voyager system [113] uses the generation capability of their TINA parser [86] to create a linguistically constrained bigram language model which is then integrated into the speech search.

1.5.3. Adaptive and Self-learning Approaches

There has also been some research into methods that allow a language processing system to self-adapt to language that is outside the coverage of its original language or grammar, or even to acquire linguistic and grammatical knowledge automatically.

CHAMP, a parsing system developed by Jill Fain Lehman in her PhD thesis [59], is a system that can adapt itself to the types of expression and linguistic idiosyncrasies of a particular user. The premise behind the approach is that the language of a user is idiosyncratic but also self-bounded, and therefore, after a period of adaptation, the system can adjust itself to the the user's preferred subset of the language. When confronted with input that is beyond the coverage of the existing grammar, CHAMP's bottom-up parser conducts a search for a "least-deviant" analysis, an analysis of a grammatical input that is closest in meaning to the original input. After a "resolution phase" in which the meaning of the utterance is established unambiguously by some interaction with the user, the system adapts its grammar and/or lexicon so that future similar expressions will be analyzable as completely grammatical. The types of deviations considered by the system include substitutions, insertions, deletions and transpositions of words. The CHAMP system has, however, some major limitations. The interpretation of deviations is heavily dependent on a detailed and accurate domain model, which limits the system's applicability to narrow domains, and hinders on its transportability to new domains. Although not designed to handle the particular types of disfluencies typical of spontaneous speech, it is unlikely that CHAMP will be suitable for such input, since only very limited amounts of deviation are tolerated by the system (due to search feasibility). Furthermore, proper adaptation in the system requires interaction with the user, which is unacceptable or undesirable in many spoken language applications.

There have also been a few attempts to apply connectionist approaches to the analysis of spoken language. This direction seems promising, since neural networks are tolerant of noisy input, and with adequate training can often automatically learn the knowledge necessary for their proper functionality. A notable work along these lines is PARSEC [36], a connectionist system developed by Ajay Jain, which is specifically designed to parse spoken language. The system consists of a number of tools for creating connectionist parsing networks that learn to parse from exposure to training sentences and their desired parses. The networks are structured, modular and hierarchically organized and are augmented with non-connectionist mechanisms for processing symbolic input. The parsed output is in the form of a case-based semantic interpretation of the input. PARSEC can tolerate some speech disfluencies, grammar deviations and speech recognition errors, with only gradual degradation in performance. It also allows easy integration of other input modalities such as prosodic information. However, the system is weak in selecting among competing analyses or speech hypotheses, and is limited by its fixed architecture to processing input of limited length and grammatical complexity.

Chapter 2

Generalized LR Parsing

2.1. Principles of LR Parsing

The Generalized LR parsing algorithm is a parsing framework that evolved out of the traditional LR parsing techniques. These techniques were originally developed for parsing programming languages in the late 1960s and early 1970s [47] [48] [17] [18]. LR parsers parse the input bottom-up, scanning it from left to right, and producing a rightmost derivation. They are deterministic and extremely efficient, being driven by a table of parsing actions that is pre-compiled from the grammar.

The common core of all of the LR parser variants is a basic Shift-Reduce parser, which is fundamentally no more than a finite-state pushdown automaton (PDA). The parser scans a given input left to right, word by word. At each step, the LR parser may perform precisely one of the following four types of actions:

1. **Shift** the next input word onto the stack, and move into a new state.

2. **Reduce** the current stack according to a grammar rule. Symbols on the stack that correspond to the right-hand side of the rule are popped from the stack, and are replaced by a symbol corresponding to the left-hand side of the rule. The parser then moves into a new state.

3. **Accept** the input.

4. **Reject** the input.

An action table that is pre-compiled from the grammar guides the LR parser when parsing an input. The action table specifies the next action that the parser must take, as a function of its current state and the next word of the input.

The stack of an LR parser is a linear structure of elements, with a defined top and bottom, where access is permitted to the top element in the stack only. It contains two types of elements: *state* nodes and *symbol* nodes. The bottom element in the stack is a state node that corresponds to the initial state of the parser. When the parser performs a shift action, a symbol node is created for the word being shifted and is pushed onto the stack. A state node corresponding to the new parser state is also created and pushed on top of the symbol node. When the parser performs a reduce action, the appropriate elements that correspond to the right-hand side of the rule being reduced are popped from the stack. The left-hand symbol of the rule is then pushed onto the stack, followed by the new state. During parse time, the state node at the top of the stack at each point in time is known as the *active* state of the parser.

Given a grammar, algorithms of the LR parsing framework automatically construct the table of parsing actions appropriate for the grammar. However, only a subset of context-free grammars (CFGs) allow the construction of a deterministic LR parsing table. The class of such grammars is called *LR grammars*, and the languages which they define are known as *LR languages*. Grammars and languages that are not LR cannot be parsed deterministically by an LR parser. The parsing table constructed for such grammars has entries that contain multiple conflicting actions. Extended lookaheads into the input can help resolve the table conflicts in certain grammars and enable the construction of a deterministic parsing table. Parsers that employ such lookaheads are called *LR(k) parsers*, and the grammars that may be parsed by them are called *LR(k) grammars*. A detailed description of the theory behind the various LR parsing techniques may be found in Aho and Ullman [4]. A more practical description of LR parsers appears in Aho and Ullman [5] and Aho et. al. [3]. A general survey of LR parsing can be found in Aho and Johnson [2].

The extreme efficiency of the LR parser is a direct result of the ability to deterministically parse the input in a single left-to-right pass. LR parsers parse their input in time that is linear in the length of the input string. LR parsers also have the *valid prefix* property, which allows them to detect ungrammatical input as early as possible.

The class of LR grammars excludes any ambiguous grammars. Thus, the ambiguity that is inherent in most grammatical descriptions of natural language appeared to imply that the efficient LR parsing techniques could not be applied to the parsing of natural languages. Other algorithms suitable for parsing general CFGs were thus used for parsing with natural language grammars. In particular, Earley's algorithm [19], Chart Parsers [43], and the CKY parsing algorithm [112] received wide-spread use and attention.

2.2. The GLR Parsing Algorithm

Tomita's Generalized LR parsing algorithm [94] extended the original LR parsing algorithm to the case of non-LR languages, where the parsing tables contain entries with multiple parsing actions. The algorithm deterministically simulates the non-determinism introduced by the conflicting actions in the parsing table by efficiently pursuing in a pseudo-parallel fashion all possible actions. The primary tool for performing this simulation efficiently is the use of a *Graph Structured Stack* (GSS). Two additional techniques, *local ambiguity packing* and *shared parse forests*, are used in order to efficiently represent the various parse trees of ambiguous sentences when they are parsed. The following subsections provide some further detail about these three major components of the algorithm. Although the original GLR parsing algorithm as designed by Tomita has a worst case time complexity of $O(n^{p+1})$, where p is the length of the the longest production in the grammar (a revised version by Kipps runs in $O(n^3)$ time), in practice the algorithm usually performs in almost linear time. Thus the performance of GLR often surpasses that of other natural language parsing algorithms.

2.2.1. The Graph Structured Stack

The *Graph Structured Stack* (GSS) is a data structure that supports the compact representation of a set of stacks of the LR parser. Similar to an LR stack, the GSS contains state nodes and symbol nodes. A state node that corresponds to the initial state of the parser is at the root of the GSS. Each

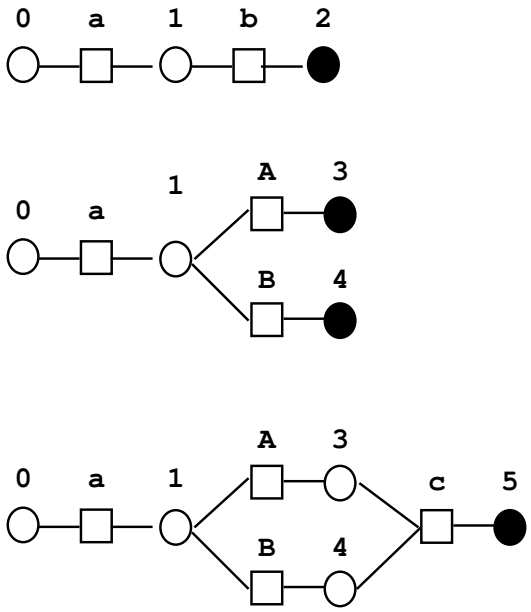


Figure 2.1 Several Instances of the GSS

path from the root of the GSS to a leaf node corresponds to an LR stack. Thus, instead of a single active state node, the GSS contains a set of active nodes.

Whenever the GLR parser encounters a situation in which the parsing table specifies multiple actions, the GSS is split into separate branches, each of which pursues one of the actions specified by the parser. Branches of the GSS are merged back together whenever they end in identical active states. The merging of such branches allows pursuing further actions of the parser that are common to these branches in an efficient way.

Figure 2.1 depicts several instances of a GSS in which these situations may be seen. State nodes are displayed as circles, with the state indicated on top of the node. Active state nodes are shadowed, and inactive state nodes are displayed as clear outlined circles. Symbol nodes are displayed as squares, with the symbol indicated on top of the node. The GSS on the top of the figure shows a case where the parser has not yet encountered a multiple action. The GSS thus consists of a single path. The GSS in the middle of the figure shows a case where the GSS was split after state 1, due to multiple actions. Two different non-terminals, A and B, where recognized, with the parser moving into states 3 and 4 respectively. The GSS on the bottom of the figure corresponds to the case where the parser has shifted the next input word "c" from both state 3 and state 4. The resulting state in both cases is state 5, and the two branches of the GSS are thus merged back together.

2.2.2. Local Ambiguity Packing

When a portion of the input sentence can be parsed and reduced to a non-terminal symbol of the grammar in more than one way, we have a local ambiguity. In the GSS, local ambiguities correspond to situations where the following conditions occur:

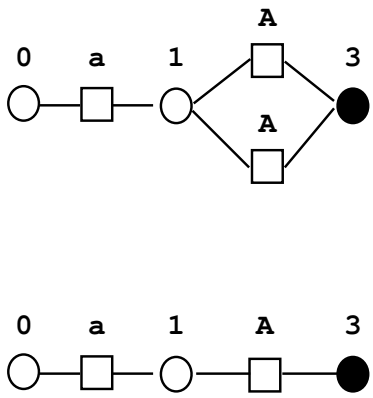


Figure 2.2 A GSS Before and After Local Ambiguity Packing

1. The GSS contains several distinct branches that start and end in the same state nodes
2. Each of the branches contains a single distinct symbol node
3. The symbol nodes all correspond to the same non-terminal symbol

An example of such a situation can be seen in the GSS on the top of Figure 2.2. The process of *local ambiguity packing* allows the parser to collapse such local ambiguities in the GSS into a single branch. The data structure that is associated with the symbol node on this branch contains pointers to the various parse ambiguities that were packed into the node. The result of a packing of the local ambiguities seen in the top GSS of Figure 2.2 can be seen in the GSS at the bottom of the figure. Local ambiguity packing increases the efficiency of the parser, because later actions that involve the packed ambiguities can be performed on the single packed branch of the GSS, instead of on each of the individual branches that existed prior to the ambiguity packing.

2.2.3. Shared Packed Forests

The GLR parsing algorithm keeps track of the parse trees constructed in the course of the parsing process via additional structures called *parse nodes*. Whenever a symbol node is created, an associated parse node is created along with it. Elementary parse nodes are created for symbol nodes of terminal symbols (as a result of shift actions). When a symbol node is created as a result of a reduction according to some rule of the grammar, a more complex parse node is constructed. The parser first obtains the symbol nodes that correspond to the right-hand side of the rule being reduced from the GSS. Subsequently, pointers to the parse nodes associated with these symbol nodes are obtained, and are then saved in a *"sons"* field of the new parse node that is created for the left-hand side symbol node. At the end of the parsing of the input, the parser can reconstruct the various parse trees obtained by following the "son" pointers in the parse nodes.

By representing the parse trees via parse nodes and pointers, the parser constructs a representation where subtrees that are common to several different parse trees are shared and represented only once. This property is called *subtree sharing* and a parse forest with such a property is called a *shared forest*.

Local ambiguity packing reduces the representation of the parse forest even further. When local ambiguity packing creates a packed symbol node, a packed parse node is created along with it. The packed parse node contains "son" pointers to the parse nodes of the various local ambiguities. A parse forest with both subtree sharing and local ambiguity packing is called a *packed shared forest*.

2.3. Computational Complexity and Performance of the GLR Parser

As noted earlier, the primary motivation in the development of the GLR parsing algorithm was the extreme efficiency of the LR parsing algorithms. Although the GLR parsing algorithm can handle any CFG[1], it is inherently designed in a way that makes it more efficient for grammars for which the pre-compiled parsing tables contain fewer multiple entries. This property can be viewed as a measure of how "close" the grammar is to being LR. It is generally the case that less ambiguous grammars translate into parsing tables with fewer multiple actions. The practical value of the GLR parser therefore depends on the nature of the grammars for which it is used. Later in this section, we mention several performance evaluations that demonstrate how the GLR parser outperforms other parsing algorithms when applied to various practical natural language grammars.

From a theoretical computational complexity standpoint, the worst case time complexity of Tomita's original version of the GLR parsing algorithm is in fact inferior to that of other common parsing algorithms. The fundamental LR(k) parsing algorithms have a time complexity of $O(n)$, where n denotes the length of the input. In contrast, common parsing algorithms for general CFGs such as Earley's have a time complexity of $O(n^3)$. The algorithm for CFL recognition with the best asymptotic time complexity known to date is due to Valiant [99], who reduced the problem of CFL recognition to that of matrix multiplication. Although the upper bound on matrix multiplication stands today at $O(n^{2.376})$ [15], the constant factors associated with Valiant's indirect algorithm make it far less attractive in practice.

Johnson [39] and Kipps [45] show that the original GLR parsing algorithm as designed by Tomita has a worst case time complexity of $O(n^{p+1})$, where p is the length of the the longest production in the grammar. Kipps, however, presents a revised version of GLR that runs in $O(n^3)$ time.

Actual performance evaluations by Tomita and others has shown that many practical grammars for the analysis of natural language are sufficiently "close" to being LR, making the use of the GLR parser preferable to that of other parsing algorithms. Tomita [94] compared the performance of the GLR parser with that of a similar implementation of Earley's algorithm. The evaluation was conducted using several different practical grammars for natural language analysis. The GLR parser outperformed the Earley parser by a factor of at least two in all of the experiments. Tomita's evaluations also demonstrated that for the various practical natural language grammars, both parsing time and space remain tractable as a function of sentence length, sentence ambiguity and grammar size.

Another comparative evaluation was conducted by Shann [87]. In his experiments, Shann compared Tomita's GLR parser with four different chart parsers with different rule invocation

[1]Tomita's original version of GLR is incapable of handling infinitely ambiguous and cyclic grammars. A later version of the parser by Nozohoor-Farshi [76] extended the algorithm to handle these cases.

```
(<DECL> <--> (<NP> <VP>)
        (((x2 agr) = (x1 agr))
         ((x0 subject) = x1)
         ((x2 form) = *finite)
         (x0 = x2)))
```

Figure 2.3 An English Grammar Rule of the GLR Parser/Compiler

strategies. The conclusions of the evaluation were that, in most cases, the GLR parser performed better than the other parsers, particularly in situations of high ambiguity. Shann contributes this result to the efficiency of the shared packed parse forest representation.

2.4. GLR and Unification Based Grammars

Much attention has been given in the last decade to unification based formalisms such as LFG [40], GPSG [23] and HPSG [79], as declarative theories for describing the structure of natural language. A fundamental primitive of these formalisms is the unification of feature based representations. Several recent practical systems for structural analysis of natural language have been designed to support unification based grammar specifications within a general CFG framework, that allows the system to incorporate known CFG parsing algorithms as the core of the system's implementation. Two examples of such systems are the Core Language Engine (CLE) [6] and the Alvey Natural Language Tools (ANLT) [11].

The Generalized LR Parser/Compiler [96] [95] is a unification based practical natural language system that was designed based on the GLR parsing algorithm at the Center for Machine Translation at Carnegie Mellon University. The system supports grammatical specification in an LFG framework, that consists of context-free grammar rules augmented with feature bundles that are associated with the non-terminals of the rules. For each context-free rule, the grammar formalism allows the specification of the feature structure associated with a left-hand side non-terminal of the rule as a function of the feature structures that are associated with the non-terminals on the right-hand side of the rule. The parser computes the actual feature values of a particular left-hand side non-terminal as a by-product of the reduce operation that reduces the right-hand side of the rule to the left-hand side non-terminal. Feature structure computation is, for the most part, specified and implemented via unification operations. This allows the grammar to constrain the applicability of context-free rules. A reduction by a context-free rule thus succeeds only if the associated feature structure unification is successful as well.

An example of an augmented grammar rule is shown in Figure 2.3. The variable $x0$ in the feature specification part of the rule is bound to the left-hand side non-terminal `DECL` of the context-free part of the rule. The variables $x1$, $x2$, etc. are respectively bound to the right-hand side constituents of the rule.

The feature structures that are computed at parse-time are attached to their corresponding parse node structures. At the end of parsing an input sentence, the parser returns a list of the top parse

nodes in the parse forest, as well as the computed feature structure associated with each of these nodes.

The Generalized LR Parser/Compiler is implemented in Common Lisp, and has been used as the parsing component of several different projects at the Center for Machine Translation at CMU in the course of the last several years.

Chapter 3

The GLR* Parsing Algorithm

3.1. Introduction

The GLR parser described in the previous chapter, as well as most other natural language parsing algorithms, are designed to analyze "clean" grammatical input. By the definition of their recognition process, these algorithms are designed to detect ungrammatical input at the earliest possible opportunity, and to reject any input that is found to be ungrammatical in even the slightest way. This property, which requires the parser to make a complete and absolute distinction between grammatical and ungrammatical input, makes such parsers fragile and of little value in many practical applications, where completely grammatical input is the exception more than the rule.

The GLR* parsing algorithm, which is described in this chapter, is an extended version of the Generalized LR parsing algorithm that was designed to be robust to two particular types of extra-grammaticality: noise in the input, and limited grammar coverage. Both phenomena cause a common situation, where the input contains words or fragments that are unparsable. The distinction between these two types of extra-grammaticality is based to a large extent upon whether or not the unparsable fragment, in its context, can be considered grammatical by a linguistic judgment. This distinction may indeed be vague at times, and of little practical importance.

GLR* attempts to overcome these forms of extra-grammaticality by ignoring the unparsable words and fragments and conducting a search for the maximal subset of the original input that is covered by the grammar. Identifying the words and fragments that the parser should ignore is not a trivial task, since the incremental nature of the parsing process is such that words and fragments that are locally grammatical may fail to combine with other parts of the input into a full grammatical sentence.

There are two major underlying assumptions behind this approach:

1. **The parse that can be obtained by using GLR* is valuable**. This assumes that the words and fragments that have to be skipped in the process of parsing the input are, for the most part, not crucial to the general correctness and utility of the obtained parse.

2. **A parse that covers more words of the input is better**. This assumption dictates that the parser should always prefer the parse that obtains the maximal coverage of the input.

Although these two assumptions form the premise upon which GLR* is founded, it is crucial that the developed parser include the proper mechanisms for identifying the conditions in which they break down. GLR* addresses these issues via a sophisticated parse scoring mechanism that allows the parser to evaluate the relative value of different alternative parses, and to identify cases when even the parse that was ranked best is not adequate. Furthermore, the parser's scoring mechanism

provides a tool by which the general preference for maximal coverage may be combined with other preferences into a general scheme for selecting the best parse from among the set of alternatives found by the parser. The scoring mechanism is addressed in full in Chapter 5.

Under the assumption that the goal of the parser is to search for the parse that obtains the maximal coverage of the original input, the problem can be formalized in the following way:

Given a context-free grammar G and a sentence S, find and parse S' - the largest subset of words of S, such that $S' \in L(G)$.

A naive approach to this problem is to exhaustively list and attempt to parse all possible subsets of the input string. The largest subset can then be selected from among the subsets that are found to be parsable. Such an algorithm is clearly computationally infeasible, since the number of subsets is exponential in the length of the input string. However, the number of *grammatical* subsets of a given input string is likely to be much smaller[1]. The idea behind GLR* is thus to devise an efficient method for parsing this collection of grammatical subsets.

3.2. The Unrestricted GLR* Parsing Algorithm

In this section, we provide an outline of the basic unrestricted GLR* parsing algorithm. This algorithm is designed to find and parse *all* possible grammatical substrings of a given input. We refer to this basic version of GLR* as "unrestricted", because the algorithm contains no restrictions on the combinations of words of the input that are allowed to be skipped. As an extension of the GLR parsing algorithm described in the previous chapter, GLR* uses similar data structures. Particularly, GLR* uses the same parsing tables that are pre-compiled from the grammar. It is only the run-time parsing part of the algorithm that is different.

The key difference between GLR and GLR* is in their use of the Graph Structured Stack. GLR uses the GSS as an extended version of a stack, where only the active nodes at the top of the stacks are accessible. GLR*, on the other hand, views the GSS as a chart that records partial parses and the state of the parser when they were created. GLR* thus maintains a structure representing the complete GSS throughout the parsing process, and allows access to all nodes in this structure, not merely the active ones.

GLR* pursues the parses of all possible subsets of the input by allowing the parser to skip over words of the input in a controlled way. With an LR style parser that uses lookaheads into the input, allowing the parser to skip over words has implications on the lookaheads used by the parser, which complicates the algorithm considerably. However, a number of studies have demonstrated that with practical natural language grammars, using a GLR parser with extended lookaheads can in fact damage the parser's efficiency [8]. For these reasons, GLR* was designed to work with no lookaheads. It therefore uses LR(0) parsing tables.

3.2.1. Design Considerations

There are two ways in which the skipping of words of the input can be implemented in the parser, a forward looking technique and a backward looking technique. With the forward looking technique,

[1]There exist CFGs for which the number of grammatical subsets of an input is still inherently exponential. However, the number of grammatical subsets in practical natural language grammars is usually much smaller.

whenever the parser moves to the processing of a new input word, it must consider the option of omitting the word. Since reduce operations in an LR(0) parser do not depend on any lookahead, it is only the shift actions of the parser that are affected. In the forward looking technique, at each new shift stage, the parser would have to consider shifting all remaining words in the input buffer from the current set of active state nodes. By shifting a word further down the input, the parser would in effect be skipping all words between the previously processed word and the new word being shifted.

The backward looking technique accommodates skipping words of the input string by allowing shift operations to be performed from inactive state nodes in the GSS. Shifting an input symbol from an inactive state is equivalent to skipping the words of the input that were encountered after the parser reached the inactive state and prior to the current word that is being shifted. Since the parser is LR(0), previous reduce operations remain valid even when words further along in the input are skipped, since the reductions do not depend on any lookahead.

The main difference between the two techniques is in execution order, namely at what point of the algorithm are word skipping shift operations considered. The forward looking technique requires the parser at each stage to process all remaining words of the input, whereas the backward looking technique uses the inactive state nodes of the GSS. A more careful inspection reveals, however, that the two methods may differ significantly in efficiency. In the forward looking technique, since the same word is shifted at different stages of the algorithm, reduce actions that follow each such shift are performed multiple times. In the backward looking technique, the word is shifted from all possible states at the same time, and subsequent reduce operations are then performed only once. GLR* therefore uses the backward looking method.

The backward looking method allows the parser to consider parses of only grammatical subsets of the input. Even though the algorithm attempts to shift each new input word from all existing active and inactive state nodes in the GSS, such a shift operation succeeds only when the partial parse that ends in a particular state node can indeed be extended by the new shifted word according to the grammar. Thus, the parser will parse all possible grammatical subsets of the original input, all of which will be represented in the final GSS.

3.2.2. Outline of the Unrestricted GLR* Algorithm

We can now present a high-level outline of the unrestricted GLR* parsing algorithm. Similar to the GLR algorithm, GLR* parses the input in a single left-to-right scan of the input, processing one word at a time. The processing of each input word is called *a stage*. Each stage consists of two main *phases*, a *reduce phase* and a *shift phase*. The reduce phase always precedes the shift phase. The outline of each stage of the algorithm is shown in Figure 3.1. `RACT(st)` denotes the set of reduce actions defined in the parsing table for state st. Similarly, `SACT(st,x)` denotes the shift actions defined for state st and symbol x, and `AACT(st,x)` denotes the accept action.

The last stage of the algorithm, in which the end-of-input symbol "$" is processed, is somewhat different. The `READ`, `DISTRIBUTE-REDUCE` and `REDUCE` steps are identical to those of previous stages. However, a `DISTRIBUTE-ACCEPT` step replaces the `DISTRIBUTE-SHIFT` step. In the `DISTRIBUTE-ACCEPT` step, accept actions are distributed to all state nodes st in the GSS (active and inactive), for which `AACT(st,$)` is true. Pointers to these state nodes are then collected into a list of final state nodes. If this list is not empty, GLR* accepts the input, otherwise, the input is rejected. Finally, the `GET-PARSES` step creates a list of all symbol nodes that are direct

```
(1) READ:
    Read the next input token x.

(2) DISTRIBUTE-REDUCE:
    For each active state node st, get the set of reduce actions
    RACT(st) in the parsing table, and attach it to the active
    state node.

(3) REDUCE:
    Perform all reduce actions attached to active state nodes.
    Recursively perform reductions after distributing reduce
    actions to new active state nodes that result from previous
    reductions.

(4) DISTRIBUTE-SHIFT:
    For each state node st in the GSS (active and inactive),
    get SACT(st,x) from the parsing table.
    If the action is defined, attach it to the state node.

(5) SHIFT:
    Perform all shift actions attached to state nodes of the GSS.

(6) MERGE:
    Merge active state nodes of identical states into a single
    active state node.
```

Figure 3.1 Outline of a Stage of the Unrestricted GLR* Parsing Algorithm

descendents of the final state nodes. These symbol nodes represent the set of complete parses found by the parser, and contain pointers to the roots of the parse forest.

3.3. Enhanced Local Ambiguity Packing

As described in the previous chapter, a local ambiguity is a situation in which a portion of the input sentence can be parsed and reduced to a non-terminal symbol of the grammar in more than one way. The GLR parsing algorithm identifies situations in which local ambiguities occur and creates a common non-terminal symbol into which the local ambiguities are packed.

Due to the word skipping behavior of the GLR* parser, local ambiguities occur on a much more frequent basis than before. In many cases, a portion of the input sentence may be reduced into a particular non-terminal symbol in many different ways, when considering different subsets of the input that may be skipped. The original local ambiguity packing procedure would have packed all such local ambiguities into the common symbol node representing the non-terminal.

Since the ultimate goal of the GLR* parser is to produce maximal (or close to maximal) parses, the process of local ambiguity packing can be used to discard partial parses that will surely not lead to the desired maximal parse. Local ambiguities that differ in their coverage of an input segment can be compared, and when the word coverage of one strictly subsumes that of another, the subsumed ambiguity can be discarded. This is due to the fact that the following *property of compositionality* holds in this case:

Property 1 *Property of Compositionality:* Let A_1 and A_2 be two local ambiguities that are packed into some non-terminal symbol A, and assume that A_1 strictly subsumes A_2 in terms of their covered words (i.e. A_2 skips more words than A_1 and every word w that was skipped in the analysis corresponding to A_1 was also skipped in the analysis corresponding to A_2). Now let T_2 be a full parse tree which incorporates the sub-analysis A_2. We now look at the parse tree T_1, which is obtained by replacing the sub-tree A_2 in T_2 with the sub-tree A_1. T_1 is a full parse tree that is guaranteed to have greater word coverage than T_2. Thus, A_2 cannot be part of a maximal parse tree, and may be discarded.

The discarding of local ambiguities that are subsumed in word coverage by other sub-analyses simplifies the later task of finding and selecting the "best" analysis, that corresponds to the maximal coverage parse, by greatly reducing the number of such different analyses produced along the way.

Due to these benefits, the local ambiguity packing process of GLR* has been enhanced to include the above mentioned sub-analysis pruning. When a new local ambiguity is found, the analyses of the two local ambiguities are compared in terms of their word coverage. In the case that the two analyses are identical in their word coverage they are packed together as they normally would have been. In the case that one of the analyses strictly subsumes that of the other, the subsumed analysis is discarded. If the two analyses differ in their word coverage, but neither one of them strictly subsumes the other, the analyses are not packed, and separate symbol nodes are maintained for each of them.

It should be noted that the unification operations that augment the unification-based version of GLR and GLR* introduces a potential problem with respect to the above described pruning process. The problem arises due to the fact that the property of compositionality is not guaranteed to hold in the case of unification augmented parsing. Because unification failures can cause rule reductions to fail, it may be the case that a subsuming sub-analysis fails to be incorporated into a final parse tree (due to a unification failure), while the subsumed sub-analysis would have succeeded, and would have eventually led to the desired maximal parse.

However, due to the large performance benefits achieved using the local ambiguity packing pruning process, we have chosen to retain the enhanced process in our current implementation of the unification-based version of GLR*. Although the described problem with the pruning process may in theory interfere with the ability of the parser to find the maximal parse, we have not observed any instances of this problem occurring in any of the actual practical applications described in this thesis.

(1) S → NP VP
(2) NP → det n
(3) NP → n
(4) NP → NP PP
(5) VP → v NP
(6) PP → p NP

Figure 3.2 A Simple Natural Language Grammar

	Reduce	Shift					Goto			
State		det	n	v	p	$	NP	VP	PP	S
0		sh3	sh4				2			1
1						acc				
2				sh7	sh8			5	6	
3			sh9							
4	r3									
5	r1									
6	r4									
7		sh3	sh4				10			
8		sh3	sh4				11			
9	r2									
10	r5				sh8				6	
11	r6				sh8				6	

Table 3.1 SLR(0) Parsing Table for Grammar in Figure 1

3.4. An Example

To clarify how the algorithm described in the previous section actually works, we present a step by step runtime example. Figure 3.2 contains a simple natural language grammar that we use. The terminal symbols of the grammar are depicted in lower-case, while the non-terminals are in upper-case. The grammar is compiled into an SLR(0) parsing table, which is displayed in Table 3.1. Note that since the table is SLR(0), the reduce actions are independent of any lookahead. The actions on states 10 and 11 include both a shift and a reduce.

To understand the operation of the parser, we now follow some steps of the GLR* parsing algorithm on the input `x = det n v n det p n`. This input is ungrammatical due to the second "`det`" token. The maximal parsable subset of the input in this case is the string that includes all words other than this "`det`".

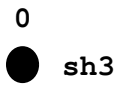


Figure 3.3 GSS Prior to SHIFT Step of Stage 1

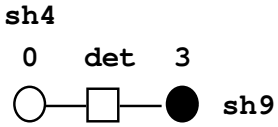


Figure 3.4 GSS Prior to SHIFT Step of Stage 2

In the figures ahead, we graphically display the GSS constructed by the parser in various stages of the parsing process. The following notation is used in these figures:

- An *active* state node is represented by a black circle with the state number indicated above it. Actions attached to the node are marked on the right.
- An *inactive* state node is represented by a clear circle. The state number is indicated above the node and attached actions are indicated above the state number.
- Symbol nodes are represented as squares. The symbol label is marked above the node.
- *Reduce* actions are denoted by **r**n, where n is the index number of the grammar rule.
- *Shift* actions are denoted by **sh**n, where n is the new state into which the parser moves after the shift action.

We follow the GSS of the parser following the reduce and shift phases of each stage of the algorithm, while processing the input string. The initial GSS contains the single active state node of state 0. Since there are no reduce actions from state 0, the first reduce phase is empty. With the first input token being "`det`", the `DISTRIBUTE-SHIFT` step attaches the action "sh3" to state node 0. Figure 3.3 shows the GSS following this `DISTRIBUTE-SHIFT` step and just prior to the SHIFT step of stage 1.

In the following `SHIFT` step, a symbol node labeled "`det`" is shifted and the parser moves into a new active state node of state 3. The algorithm then proceeds to the next stage to process the next input token "`n`". Since there are no reduce actions from state 3, the reduce phase of this stage is empty. In the following `DISTRIBUTE-SHIFT` step, shift actions are distributed by the algorithm to both the active node of state 3 and the inactive node of state 0. Figure 3.4 shows the GSS after this step, just prior to the `SHIFT` step of stage 2.

In the following `SHIFT` step, the input token "`n`" is shifted from both state nodes, creating new active state nodes of states 9 and 4. The shifting of the input token "`n`" from state 0 corresponds to a parse in which the first input token "`det`" is skipped. Stage 3 begins to process the next input token "`v`". First, in the `DISTRIBUTE-REDUCE` step, reduce actions are distributed to both

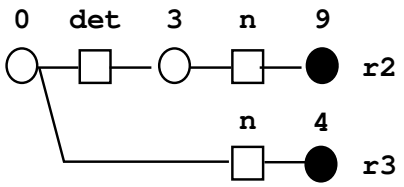


Figure 3.5 GSS Prior to REDUCE Step of Stage 3

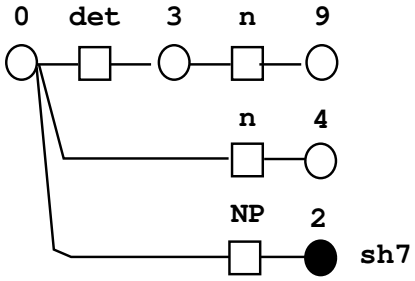


Figure 3.6 GSS Prior to SHIFT Step of Stage 3

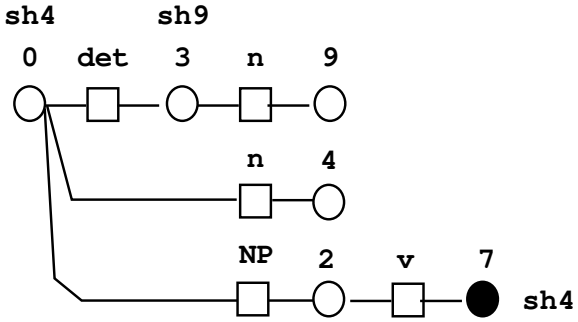


Figure 3.7 GSS Prior to SHIFT Step of Stage 4

existing active nodes. Figure 3.5 shows the GSS at this point in time, just prior to the REDUCE step of stage 3.

The following REDUCE step reduces both branches into noun phrases. The two "NP"s are packed together by the local ambiguity packing procedure. In the following DISTRIBUTE-SHIFT step, shift actions for the input token "v" are distributed to all the state nodes. However, in this case, only state 2 allows a shift of "v" (into state 7). The resulting GSS, prior to the SHIFT step of stage 3, is displayed in Figure 3.6.

The following SHIFT step shifts the word "v" and creates an active state node of state 7. The next stage begins by reading the next input token "n". The state 7 node is the only active node at this point. Since no reduce actions are specified for this state, the following reduce phase is empty. Shift actions are then distributed in the DISTRIBUTE-SHIFT step. The resulting GSS is shown in Figure 3.7.

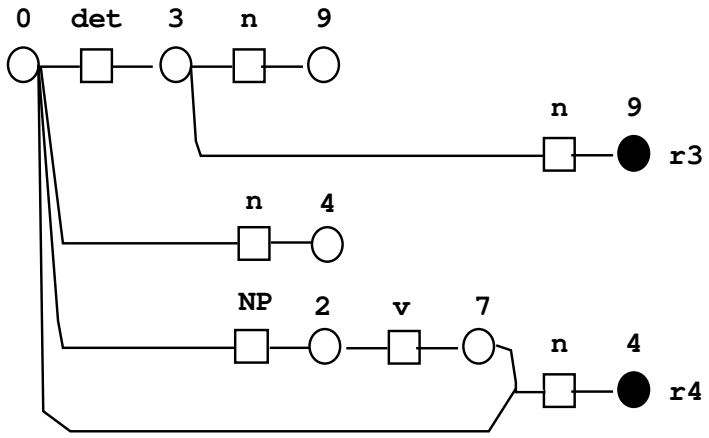


Figure 3.8 GSS Prior to REDUCE Step of Stage 5

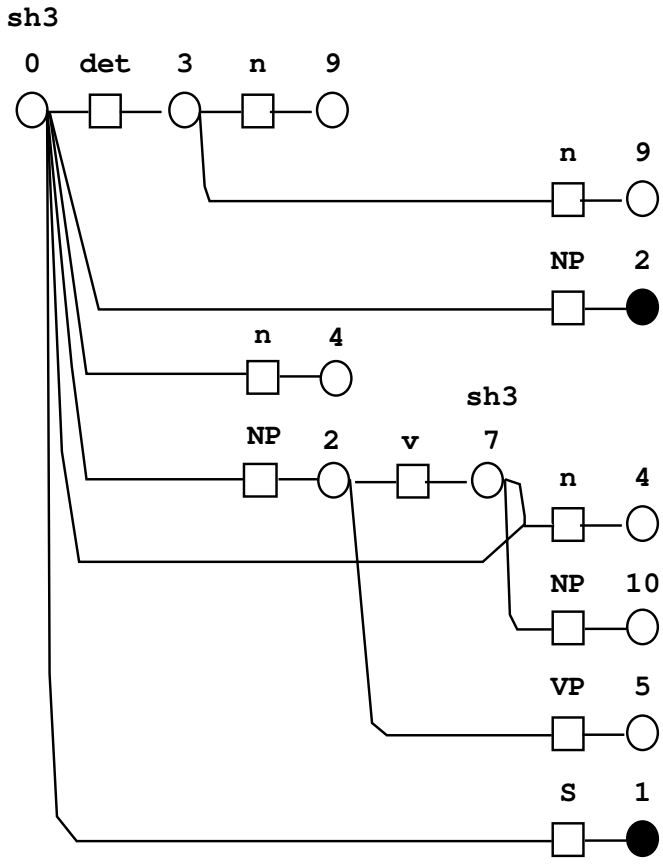


Figure 3.9 GSS Prior to SHIFT Step of Stage 5

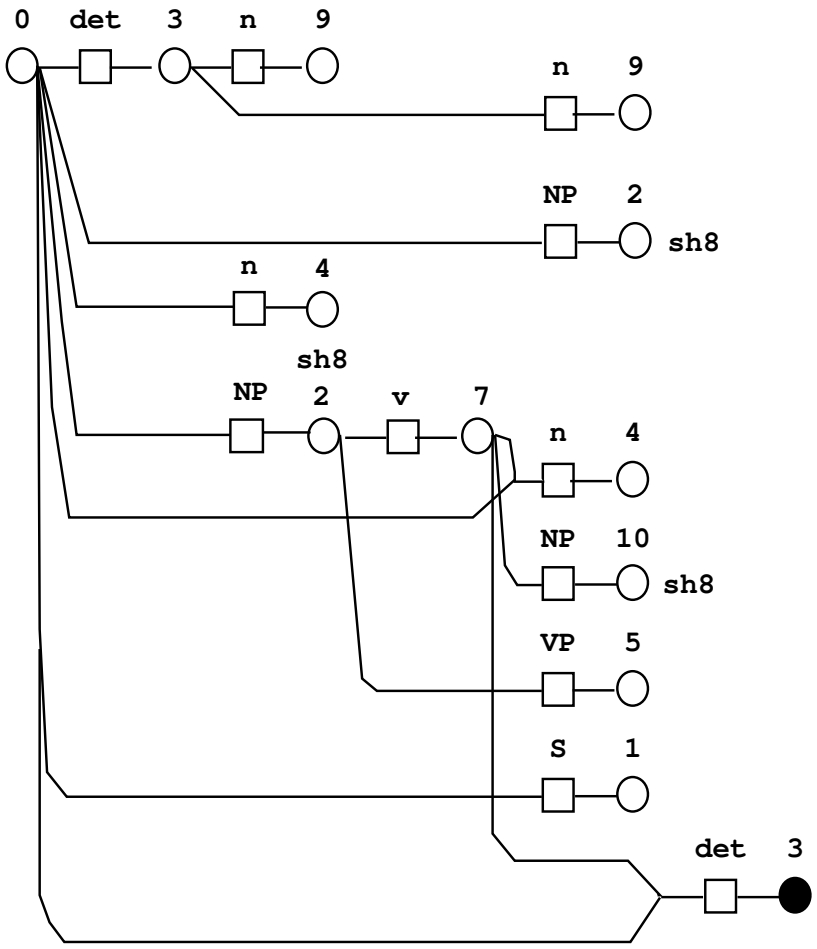


Figure 3.10 GSS Prior to SHIFT Step of Stage 6

The subsequent SHIFT step then shifts the input token "n" from the three nodes that were marked with shift actions. The next input token is "det". Figure 3.8 shows the GSS just prior to the next REDUCE step and Figure 3.9 after the REDUCE and DISTRIBUTE-SHIFT steps, just before the SHIFT step. Note that the current input token "det" cannot be shifted from either of the two active state nodes at this point. A GLR parser would have thus failed as this point. However, the GLR* algorithm succeeds in distributing the shift actions to two inactive state nodes in this case. The token "det" is then shifted from these two nodes.

The next stage then begins, and the input token "p" is read. Since no reduce actions can be distributed to the active state node 3, the reduce phase is empty. DISTRIBUTE-SHIFT then distributes shift actions to all state nodes. The GSS at this point is shown in Figure 3.10. Note that once again, shift actions could not be distributed to the active state node, since the grammar does not allow "p" to follow "det". However, a shift action was distributed to the node of state 10. This branch skips over the previous "det", and eventually leads to the best maximal parse.

For the sake of brevity we do not continue to further follow the parsing step by step. The final GSS, immediately following the DISTRIBUTE-ACCEPT step, is displayed in Figure 3.11.

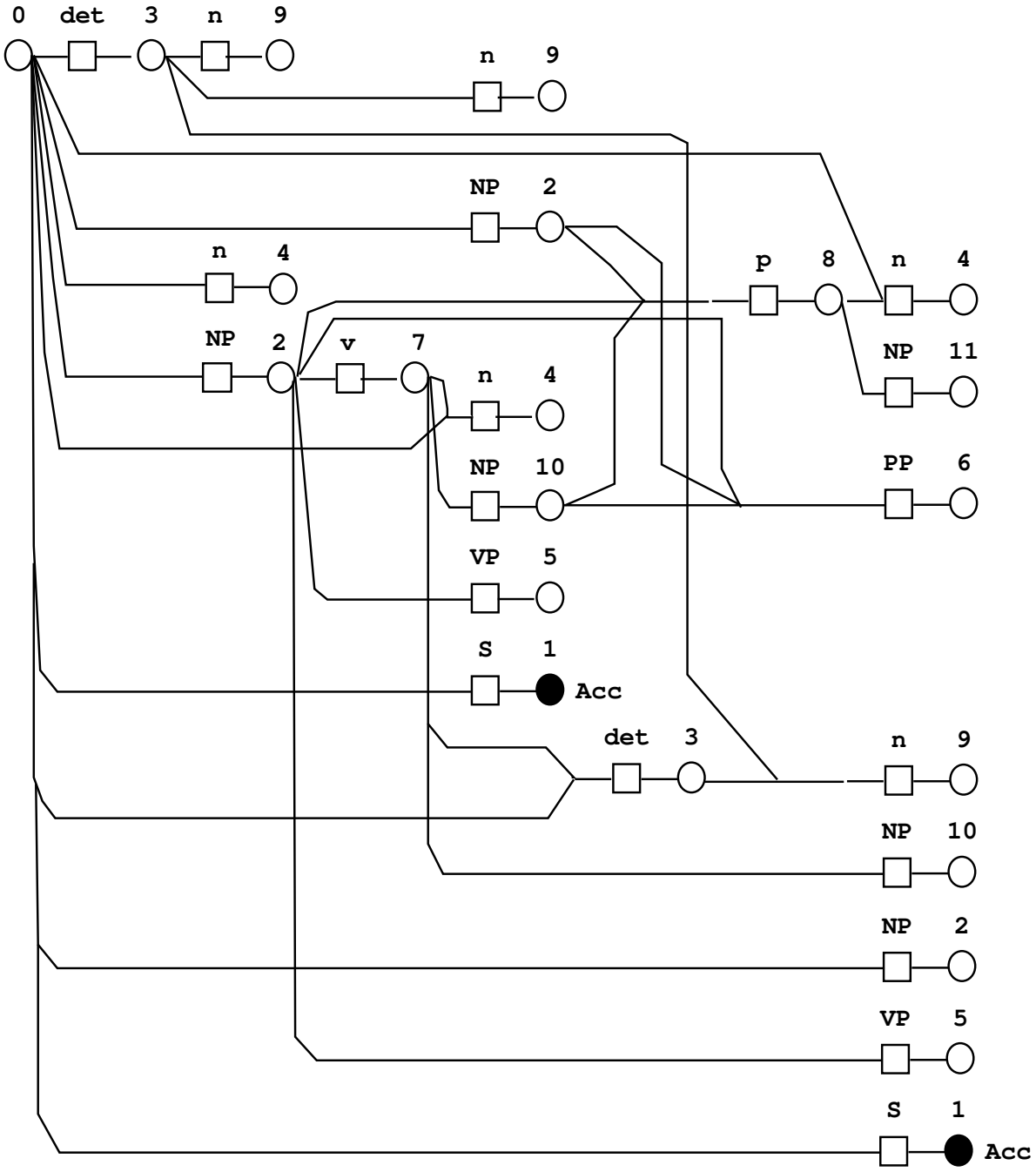


Figure 3.11 Final GSS after DISTRIBUTE-ACCEPT Step

Several different parses, corresponding to different subsets of skipped words are actually packed into the "S" symbol node seen at the bottom of the figure. The parse that corresponds to the maximal subset of the input is one in which the second "det" was the only word that was skipped.

3.5. Complexity and Performance of the Unrestricted GLR* Algorithm

In this section we look at the theoretical time and space complexities of the unrestricted GLR* algorithm, with respect to both the length of the input and the size of the grammar. We compare the complexity measures of GLR* with those of the underlying GLR parsing algorithm. However, since complexity measures reflect worst-case asymptotic behavior and disregard constant factors, they provide only limited insight into the actual time and space performance of the GLR* parser. Therefore, following the complexity analysis, we take a look at some actual performance evaluations of the unrestricted GLR* parser.

Works by Johnson [39], Kipps [45] and Carroll [11] have included thorough analyses of the computational time complexity of Tomita's GLR parsing algorithm. With respect to the input length, Kipps shows that GLR has a general time complexity of $O(n^{p+1})$, where n is the length of the input, and p is the length of the longest production in the grammar[2]. Kipps proposes a modified version of GLR for *recognition only*, with an improved time complexity of $O(n^3)$ for grammars with rules of arbitrary length. Carroll extends this modified algorithm so that parsing as well can be done in time $O(n^3)$.

With respect to the grammar size, Johnson shows that there exists a class of grammars G_m, for which GLR requires time $\Omega(c^{\sqrt{|G_m|}}\, n)$. Thus, GLR has a non-polynomially-bound time complexity with respect to the size of the grammar.

Our analysis of the time complexity of GLR* parallels the analyses of Kipps and Carroll for the GLR parsing algorithm. This allows us to compare the complexity of GLR* with that of GLR on a step by step basis.

3.5.1. Time complexity of Unrestricted GLR*

Complexity with Respect to the Input Length

We now look at the time complexity of the unrestricted GLR* parsing algorithm with respect to the length of the input. For the purpose of the analysis, we first look at a single stage of the algorithm, as it is described in Figure 3.1, and then sum up for all stages. In the following analysis, $|S|$ denotes the total number of states in the parsing table, and p denotes the length of the longest grammar rule. Most of the following analysis parallels that of Kipps [45]. Assume we are in the i-th stage (processing the i-th word of input). We look at the complexity of each step in this stage:

1. READ: Reading the next input word requires $O(1)$ time.

[2]The length of a grammar rule is the number of symbols on the right-hand side of the rule

2. DISTRIBUTE-REDUCE: The number of active state nodes is bounded by the constant $|S|$. For each active state node, we must access the appropriate state entry in the parsing table in order to determine the reduce actions. Thus, the number of such required accesses to the parsing table is bounded by $|S|$. Each such access to the parsing table can be done in constant time, and attaching the fetched reduce actions to the active state node also requires only constant time. Therefore, an entire DISTRIBUTE-REDUCE step requires only $O(1)$ time.

3. REDUCE: A REDUCE step is comprised of a series of discrete reductions. Each reduction requires only constant time, so the complexity of the entire step is equal to the number of reductions performed. The total number of reductions in a REDUCE step is the sum of the number of initial reductions from active state nodes, and the number of subsequent recursive reductions. First, we note that the "fan-in" of a state-node (i.e. the number of different paths leading into the state node) is $O(i)$. Since each reduction is bounded in length by p, the total number of reduce paths from an active state node is at most $O(i^p)$. Thus, the total number of initial reductions is at most $O(i^p)$. Kipps' analysis shows that the total number of recursive reductions in the GLR case is also at most $O(i^p)$. This holds in the GLR* case as well. Therefore, the total number of reductions is at most $O(i^p)$, as is the complexity of the entire REDUCE step.

4. DISTRIBUTE-SHIFT: Similar to the DISTRIBUTE-REDUCE step, the distribution of shift actions to an individual node requires only constant time. However, in GLR*, shift actions are distributed to all state nodes in the GSS, not merely the active state nodes. At stage i, the total number of state nodes in the GSS is $O(i)$. Thus, the time complexity of the DISTRIBUTE-SHIFT step is $O(i)$.

5. SHIFT: A single shift operation requires only constant time. Since the previous step distributed at most $O(i)$ shift actions, the total time for actually performing all distributed shift operations is $O(i)$.

6. MERGE: A single scan through all the new state nodes created in the SHIFT step is sufficient to merge all new state nodes of identical states. Thus, the time complexity of this step is $O(i)$.

From the above analysis, it can be seen that the complexity of the REDUCE step dominates the complexities of all the other steps in a single stage of the algorithm. Thus, the complexity of the i-th stage of GLR* is $O(i^p)$. Summing up for all n stages of the algorithm, we obtain that the total time complexity of the unrestricted GLR* algorithm is $O(n^{p+1})$.

One should note that for the original GLR algorithm, Carroll's improved version allows the REDUCE step to be performed in time $O(i^2)$. Carroll's modifications to GLR are completely independent of the changes that modify GLR into GLR*. Consequently, the same improvement can be applied to GLR* as well, with the resulting benefit on improving the time complexity of the REDUCE step to $O(i^2)$. However, as Carroll notes [11] (p. 111), his modification cannot be applied to a unification-based parsing algorithm, such as the unification-based versions of GLR and GLR*. Thus, our current implementation of the unification-based GLR* parser does not incorporate Carroll's improvement. Carroll also notes [11] (p. 105) that the overheads associated

with his proposed improvement may in fact negate the expected gains from the improved time bound, and the question of whether the modification would improve parsing performance for a particular grammar has to be investigated empirically. We suspect that Carroll's modifications, had they been implementable for GLR*, would have had little affect on the performance of the parser in the context of the applications investigated in this thesis.

The above complexity analysis has shown that the time complexity of GLR* with respect to input length is identical to that of the original GLR parsing algorithm. In fact, GLR* and GLR differed in complexity only in the `DISTRIBUTE-SHIFT`, `SHIFT` and `MERGE` steps, which are not the dominant components in the complexity of an entire stage. Thus, these differences became insignificant in the overall complexity of the algorithms.

At first glance, this result may seem surprising. Since GLR and GLR* both have a time complexity of $O(n^{p+1})$, the asymptotic running time of GLR* is guaranteed to be slower than that of GLR by no more than a constant factor.

However, several important points should be noted to put this complexity result in its proper context:

1. The above complexity analysis was conducted on only the *recognition* aspect of the algorithm. As it turns out, taking into account the operations required for producing parses in both GLR and GLR* does not affect the time complexity results, since neither GLR nor GLR* produce parse trees, but rather represent the set of analyzed parses in a packed parse forest. Thus, even though GLR* may find an exponential number of parsable subsets for a given input string, these parses are implicitly represented in the packed forest in a compact way.

2. The determined complexity bound is a *worst case* analysis. The result by itself does not preclude the possibility of a substantial difference in runtime between GLR and GLR* on *particular* inputs or even on average inputs with *specific* grammars. It is thus possible that while GLR is extremely efficient for particular grammars or classes of grammars, GLR* performs much worse on the same grammars.

3. Since the above analysis related runtime as a function of only the length of the input string, the "constant" ratio between the asymptotic runtime of GLR and GLR* may in fact be dependent on properties of the grammar. It is thus important to look at the complexities of both algorithms as a function of the size of the grammar as well.

Complexity with Respect to the Grammar Size

As already noted, Johnson [39] demonstrates that for a particular class of grammars G_m, there are inputs for which the GLR parsing algorithm operates in time that is $\Omega(c^{\sqrt{|G_m|}}\, n)$. This shows that GLR has in fact a non-polynomially-bound runtime with respect to the size of the grammar.

The most important way in which the grammar affects the runtime performance of both GLR and GLR* is indirectly via the number of states in the parsing table that is pre-compiled from the grammar. It is interesting to observe that the class of grammars Johnson uses in his analysis is one where the number of states in the parsing table is exponentially related to the grammar size. The class G_m is defined such that where $|G_m| = O(m^2)$, the number of states in the corresponding parsing table is $O(2^m)$. Additionally, the class of grammars that Johnson uses has the property that when parsing an input of the form a^n (for $n > m$), at the end of each stage i of the parsing (for

$i > m$), the GSS contains $\Omega(2^m)$ active state nodes. Stage i of the parsing also requires $\Omega(2^m)$ reduce operations to be performed, which results in the observed time bound.

The time bound for GLR* on this same example will, of course, be greater or equal to that found for GLR. The number of active state nodes at stage i of the algorithm remains $\Omega(2^m)$, but the total number of reduce operations performed at stage i is $\Omega(i2^n)$. This results in a total runtime of $\Omega(c^{\sqrt{|G_m|}}\ n^2)$ for this class of grammars.

Johnson's example thus shows that both GLR and GLR* have a non-polynomial runtime bound with respect to the size of the grammar, but does not provide much insight into the actual differences in runtime behavior between the two algorithms on common practical grammars. However, the analysis of the example does provide some insight on the effect that the number of active state nodes at each stage (and the number of reduce operations from these active nodes) have on the total parsing runtime. Actual runtime evaluations using practical natural language grammars, which we present in the next subsection, indeed show that while for GLR, the number of active state nodes and reduce operations per stage is on average very small, for GLR* these numbers grow very quickly. As a result the runtime performance of the two parsers quickly diverges in practice.

3.5.2. Runtime Performance of Unrestricted GLR*

The complexity analyses presented in the previous subsection provided us with very limited insight into the expected actual time and space performance of the GLR* parser. I therefore conducted experiments to evaluate the time and space behavior of the unrestricted GLR* algorithm in a practical setting.

The Evaluation Benchmark

For the purpose of evaluating the time and space performance of the GLR* parser, I selected a benchmark setting from the JANUS speech-to-speech translation project, a "real world" setting in which GLR* is currently being used. The benchmark was constructed from the JANUS English Spontaneous Speech Translation (ESST) domain, which handles spontaneously spoken English dialogs between a pair of two people attempting to schedule a meeting.

The grammar used in these evaluations is the November-94 version of the ESST GLR Grammar developed by Keiko Horiguchi for the JANUS Project. The grammar contains 805 rules which compile into a GLR parsing table of 987 states.

The data set used for the benchmark is a subset of the ESST development set data. The development set data consists of transcribed text of actual recorded dialogs, where the parties attempted to schedule a meeting. The transcribed utterances are broken into sentences. The development set was then used to incrementally construct a wide coverage English grammar for the domain, which resulted in the above mentioned grammar. The benchmark data set appears in Appendix B. It contains 552 sentences from the ESST development set. Table 3.2 shows the sentence length distribution of the benchmark data set.

Time and Space Performance of Unrestricted GLR*

Using the benchmark setting described above, I conducted an evaluation of the time and space performance of the unrestricted GLR* parser as well as the original GLR parser. Loaded with the

Sentence Length	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Number of Sentences	119	31	25	45	57	50	41	38	31	32	27	17	15	12	12

Table 3.2 Sentence Length Distribution of Benchmark Data Set

benchmark grammar, the parser was run on the entire set of 552 input sentences. The experiment was conducted on an HP-735 Apollo workstation running HP-Lucid Common Lisp.

The parser was run twice in two different configurations. In the first configuration, performance data was collected on unrestricted GLR*. Therefore, the parser was allowed unrestricted word skipping. In the second configuration, performance data was collected with the parser's word skipping disabled. This configuration is equivalent to using the original GLR parser.

To evaluate time performance, actual parse times were measured. To evaluate space performance, the total number of GSS parse nodes created in the course of parsing each sentence was measured. The parse times and node counts were then averaged for each sentence length.

The average parse times and parse node counts for each of the sentence lengths are shown in Figure 3.12 and Figure 3.13 respectively.

Discussion

As can immediately be seen from the time and space performance figures, the time and space behavior of the unrestricted GLR* parser diverges very rapidly from that of GLR, and becomes infeasible for even sentences of medium length. Whereas GLR time and space requirements increase roughly linearly as a function of the sentence length, thus achieving an almost best possible performance, the performance of unrestricted GLR* appears to be similar to that predicted by the worst case complexity analysis presented earlier in the chapter.

We may thus conclude that the unrestricted GLR* parser is infeasible to operate in actual real-world scenarios, and that severe restrictions on the word skipping behavior of the parser must be added to the parser in order to achieve acceptable levels of performance, that will allow the feasible use of GLR* in practical applications.

3.6. Controlling Parser Search

The unrestricted GLR* algorithm that we have described so far computes parses of *all* parsable subsets of the original input string, the number of which can be exponential in the length of the input string. As we discovered in the previous section, the computational cost of finding all possible parsable subsets of the input can become infeasible, when working with a reasonably large natural language grammar. However, if we are primarily interested in only maximal or close to maximal parsable subsets of the input string, it is possible to dramatically cut down the computational cost by using effective heuristic search techniques. This section describes two different heuristic search techniques that I have developed for the GLR* parser, and analyzes their effectiveness in finding the desired maximal parses, while cutting down the computational cost of running the parser.

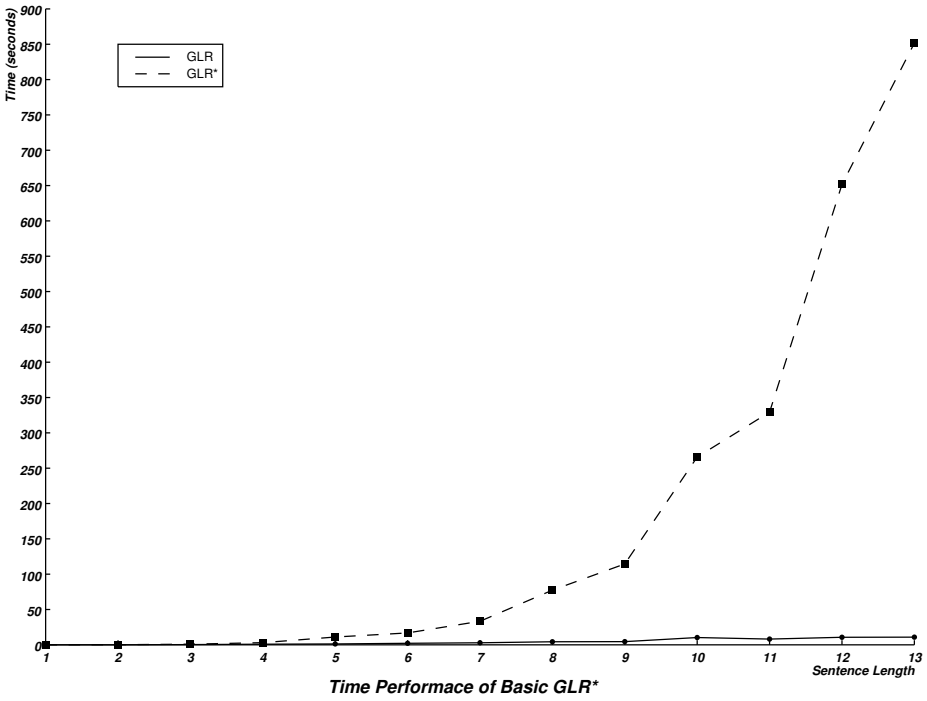


Figure 3.12 Time Performance of Unrestricted GLR* as a Function of Sentence Length

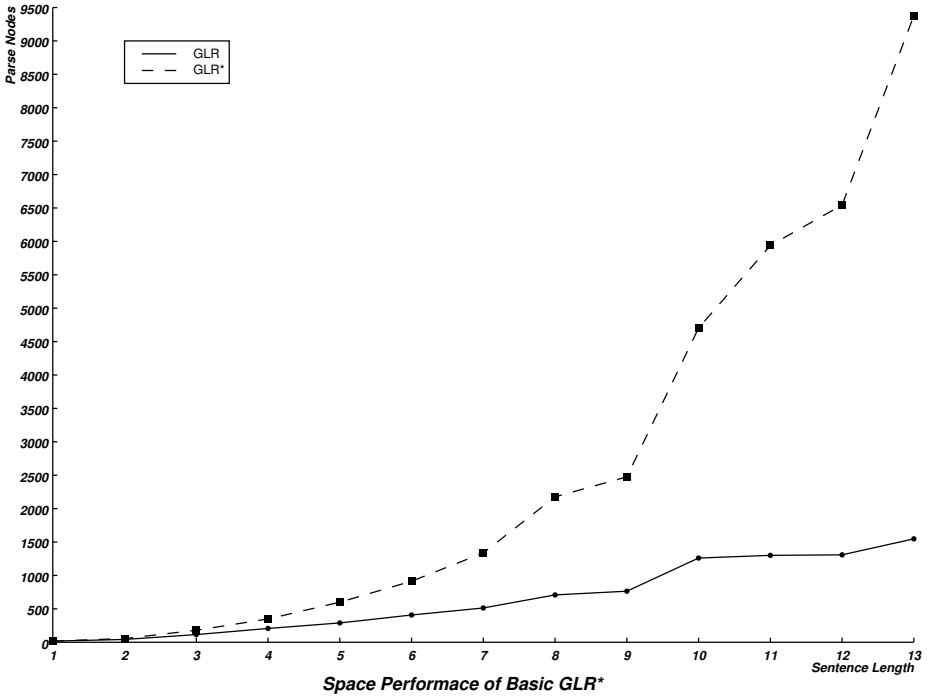


Figure 3.13 Space Performance of Unrestricted GLR* as a Function of Sentence Length

In order to develop effective search strategies for the GLR* parser, we must first define the notion of a search space in which the parser is operating. Similar to the underlying GLR parser, GLR* can be viewed as a non-deterministic parser. Whereas the non-determinism of GLR was represented in the form of the multiple actions defined in the parsing table, GLR* has an additional source of non-determinism - the choice of input words that are skipped. However, these two sources of non-determinism can be viewed in a uniform way. For each specific parse produced by GLR*, there is a unique sequence of actions that the parser performed in the process of deriving the parse. This one-to-one correspondence implies that restricting the set of *actions* that the GLR* parser performs at each step of the parsing process, corresponds to a similar restriction on the set of *parses* produced by the parser. Thus, the search for a particular parse or set of parses corresponds to a search for the particular sets of parsing actions that produce the desired parses.

This leads to the following notion of a search space. The GSS of the parser at each step of the parsing process can be viewed as a *"state"* of the search space. The various possible parsing actions can be viewed as the *"operators"* that transform the parser between states. A search strategy that operates within this notion of a search space would be one that effectively limits the set of operators (i.e. parsing actions) being applied at each step as much as possible, while still pursuing those actions that lead to a desired "goal" state (i.e. the desirable parse or set of parses).

Based on the above notion of the parser search space, I developed and experimented with two heuristic search techniques that limit the sets of parsing actions that are considered by the parser at each step of the parsing process. These search heuristics are aimed at restricting the *word skipping* behavior of the parser, and are not designed to resolve parse ambiguity. Therefore, no attempt is made to prune out actions that are a result of multiple entries in the parsing table.

The following two heuristic search techniques were developed. The details of each of the heuristic methods are described in the following subsections.

1. **The k-word Skip Limit Heuristic**: The parser is restricted to skip no more than k consecutive input words at any point in the parsing process.

2. **Beam Search Heuristic**: The parser is restricted to pursue a "beam" of a fixed size k of parsing actions. Parsing actions are selected according to a criterion that locally minimizes the number of words skipped.

3.6.1. The k-word Skip Limit Heuristic

The simpler of the two heuristics is the *k-word skip limit heuristic*. The idea behind this heuristic is to limit the parser to skipping at most k consecutive input words, where the parameter k can be set to any desired value prior to runtime.

The heuristic is implemented in the GLR* parser in a straightforward way. A "level" field is added to each state node of the GSS, representing the position in the sentence of the word being processed when the state node was created. When shift actions are distributed in the `DISTRIBUTE-SHIFTS` step, the level field of each state node is compared with the level of the current word being shifted. Shift actions are then distributed only to state nodes with a level that is of distant k or less than the current level.

The value of the parameter k can be set by the user at runtime, prior to parsing an input. Setting the value of k to 1 is equivalent to completely disallowing any word skipping, in which case GLR* behaves exactly like the original GLR parser. Setting the value of k to a value equal or greater

than n - the length of the sentence - is equivalent to disabling the heuristic, since all possible word skipping combinations will be considered.

Using this heuristic (with k set to a value less than the length of the input sentence), GLR* is no longer capable of finding all parsable subsets of the original input, since it can only find parsable subsets that do not require skipping more than k consecutive words. The heuristic is thus effective in cases where a reasonably small setting of k, which results in a significant cut in the number of shift actions distributed by GLR* in `DISTRIBUTE-SHIFT` steps, is sufficient for finding the desired maximal parsable subset of the input. Empirical results of using this heuristic are presented in a later subsection.

3.6.2. The Beam Search Heuristic

The *beam search heuristic* is a heuristic in which the parser is restricted to pursue at each step a "beam" of a some size k of parsing actions. Whenever there are more than k alternative actions, the heuristic must select the "best" k actions that are to be pursued, according to some pre-designed criterion. In the case of the GLR* parser, an appropriate criterion would be one that attempts to minimize the overall number of words skipped.

A direct way of conducting a beam search would be to limit the number of active state nodes pursued by the parser at each step, and continue processing only active nodes that are most promising in terms of the number of skipped words associated with them. However, the structure of the GSS makes it difficult to associate information on skipped words directly with the state nodes. This is due to the fact that nodes of equal states from different sub-parses are merged. Thus, a state node may be common to several different sub-parses, with different skipped words associated with each sub-parse.

We therefore use a somewhat different heuristic that has a similar effect. Since the skipping of words is the result of performing shift operations from inactive state nodes of the GSS, our heuristic limits the number of inactive state nodes to which shift actions are distributed in the `DISTRIBUTE-SHIFT` step. This is similar to the k-word-skip-limit heuristic, but the criterion for deciding which shift actions to distribute is different. With the beam search heuristic, shift actions are first distributed to the active state nodes of the GSS. This corresponds to no additional skipped words at this stage. If the number of distributed shift actions at this point is less than the preset constant k (the "beam-width"), shift actions are distributed to inactive state nodes as well. Inactive states are processed ordered by their level, so that shift actions from more recent state nodes (with higher levels) that result in fewer skipped words are considered first. Shift operations are distributed to inactive state nodes in this way until the number of shifts distributed reaches k.

The advantage of using the beam search heuristic versus the k-word-skip-limit heuristic is in it's dynamic flexibility. With the beam search, by distributing just enough shift actions in order to fill the beam, the parser dynamically considers skipping more words in situations where there are fewer possible sub-parses to pursue. If, on the other hand, at some step the distribution of shifts actions to active state nodes already fills the beam, then no words will be skipped at this point in the parsing process.

Experiments we conducted with the beam search heuristic suggested a few minor modifications to the heuristic. First, we changed the heuristic so that shift action are distributed to all possible active state nodes, even when the number of such actions exceeds the beam-width. Thus, the GLR* parser pursues at least all possible parses that do not require any skipping at all. Additionally,

whenever shift actions cannot be distributed to any active state nodes, the first level of nodes to which shift actions *can* be distributed is treated as if it were the "active" level, and shift actions are distributed to *all* possible state nodes of that level. Thus, in situations where the parser *must* skip over a sequence of words in order to be able to continue, it will consider *all* possible parses that arise from skipping this particular sequence of words.

The beam-width parameter k can be dynamically set to any constant value at runtime. Setting the beam-width to a value of 0 disallows shifting from inactive states all together, which is equivalent to using the original GLR parser. Empirical results of using the beam search heuristic are presented in the next subsection.

3.6.3. Empirical Evaluation of the Search Heuristics

In this section we evaluate the two search heuristics described above on actual data. Both described search heuristics are controlled by parameters that restricts the word skipping behavior of the parser, and thus limits the search. In the case of the k-word skip limit, the parameter is k, the maximum number of consecutive words allowed to be skipped. With the beam search heuristic, the beam-width parameter controls the amount of word skipping allowed.

There exists a direct tradeoff between the amount of search (and word skipping) that the parser is allowed to pursue and the time and space performance of the parser itself. If the control parameter is set too low, the best parse for a given input may not be found, if the search heuristic prohibits the parser from skipping the complete set of words that need to be skipped. On the other hand, if the parameter is set unnecessarily high, the parser will needlessly consider many more word skipping possibilities than is actually necessary for finding the best parse. This results in a needless slowdown in parser performance.

The goal is therefore to determine the smallest possible setting of the control parameters that still allows the parser to find the desired parses in an overwhelming majority of cases. The actual "optimal" setting of the control parameter thus depends on the actual amount of skipping that is required by the data itself. This is influenced by various characteristics of the data, such as the domain, the form of the input (i.e text, transcribed speech, output from a speech recognizer, etc.) and the amount of noise in the input. Also influential are properties of the grammar being used. For example, a larger grammar may require a higher setting of the beam-width parameter, in order to achieve some desired amount of word skipping.

Time and Space Behavior

We now look at some empirical evaluations of parser behavior with different settings of the search control parameters. Both runtime and space are considered. These evaluations were conducted using the same benchmark setting on which the unrestricted GLR* parser was evaluated, as described earlier.

Loaded with the benchmark grammar, the parser was run multiple times on the benchmark set of 552 input sentences. With each run, the parser's control parameters were set differently. We experimented with several settings of the skip word limit parameter, and with several settings of the beam-width parameter. Although it is possible to combine the effects of both control parameters simultaneously, this was avoided in this evaluation. As before, the experiment was conducted on an HP-735 Apollo workstation running HP-Lucid Common Lisp.

The time and space performance of the GLR* parser, with various settings of the skip word limit parameter, can be seen in Figure 3.14 and Figure 3.15 respectively. Similarly, the performance with various settings of the beam-width parameter can be seen in Figure 3.16 and Figure 3.17.

Discussion

As could be expected, time and space requirements of GLR* increase as the control parameters are set to higher values. This increase however is gradual, and in the case of the beam parameter, even with considerably high settings, parser performance remains in the feasible range, and does not approach the time and space requirements experienced when running the unrestricted version of GLR*.

A comparison of the performance figures of the two heuristic control mechanisms reveals the clear advantage of using the beam search, versus the simpler skip word limit heuristic. With the skip word limit heuristic, time and space requirements increase rather rapidly. Even with a setting of just one (i.e. no more than a single word can be skipped between any two parsed words), parse time increases to more than double that of the original GLR parser, on input sentences of length 10 or more. This is roughly comparable to using a beam-width value of 30. Using a skip word limit of 2 already results in parse times that are unacceptably long for sentences of length 10 and up. However, experiments in using GLR* within the JANUS project have shown that a skip word limit of at least 2 is necessary to achieve even moderate benefits from using GLR* on parsing spontaneous speech. On the other hand, using the beam search heuristic, we have achieved similar increased performance when setting the beam-width as low as 10. The flexibility that is built into the beam search heuristic that allows the parser to skip over a different number of words at various points in the parsing process is at the core of this result.

As mentioned earlier, determining the optimal setting of the beam-width parameter is a task that depends on many different factors. This optimal setting may thus differ widely from task to task. It is plausible that there are tasks where in order to achieve reasonable increased parsability from the GLR* parser, the beam-width parameter would have to be set to such high values where parser time and space performance degrades to unacceptable levels. However, in the practical applications in which we have tested the GLR* parser so far, this has not proven to be the case.

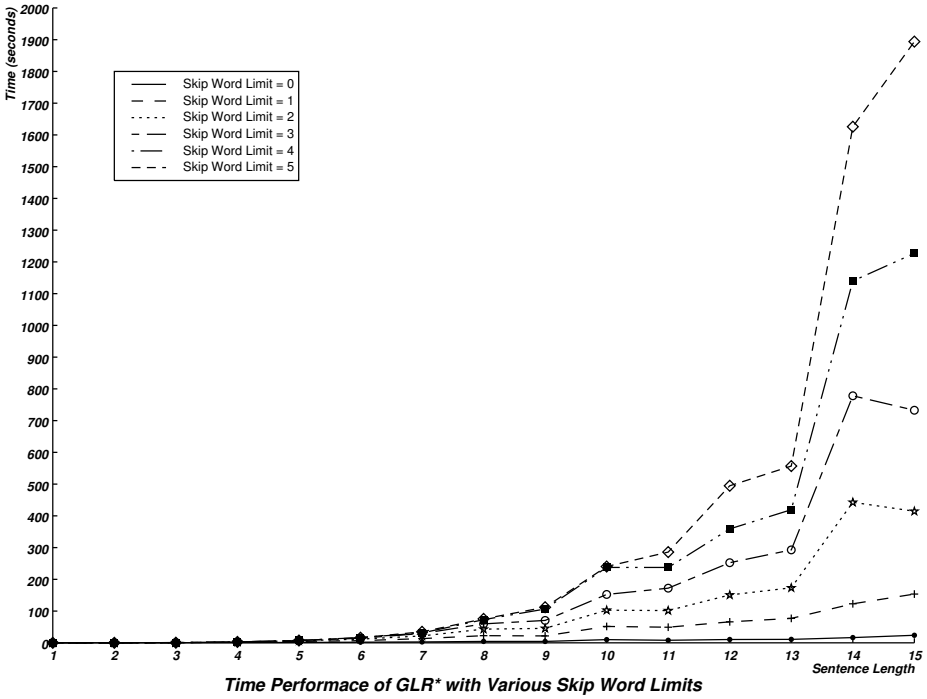


Figure 3.14 Time Performance of GLR* with Various Skip Word Limits as a Function of Sentence Length

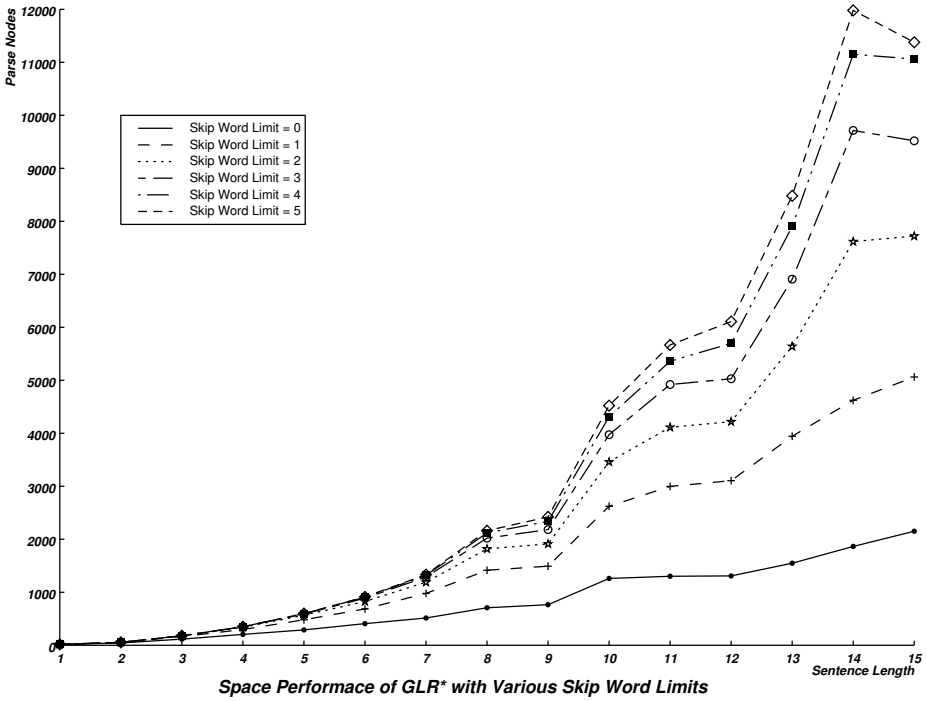


Figure 3.15 Space Performance of GLR* with Various Skip Word Limits as a Function of Sentence Length

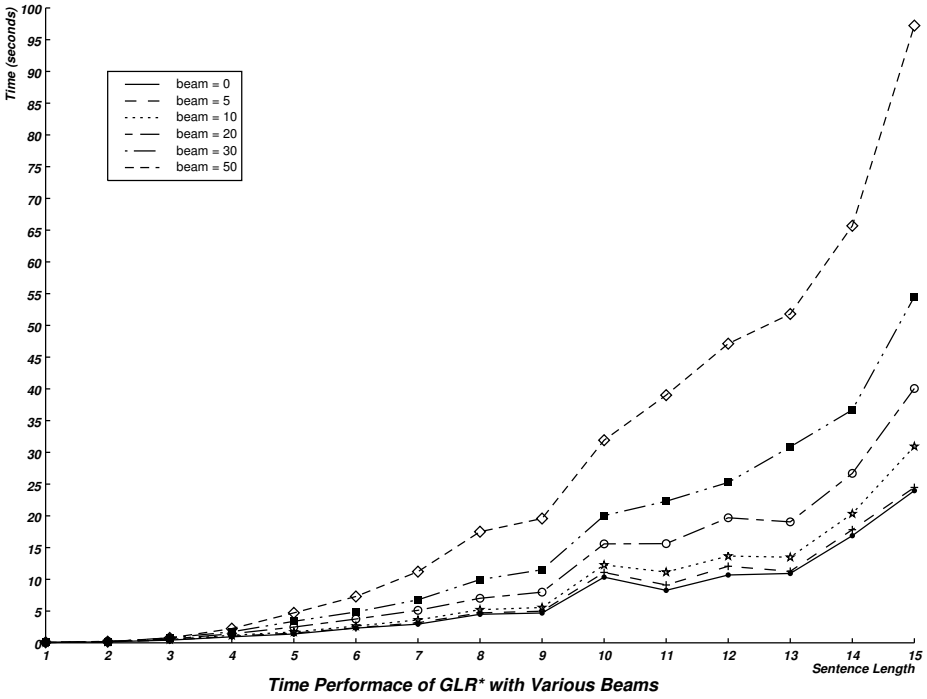


Figure 3.16 Time Performance of GLR* with Various Beam Widths as a Function of Sentence Length

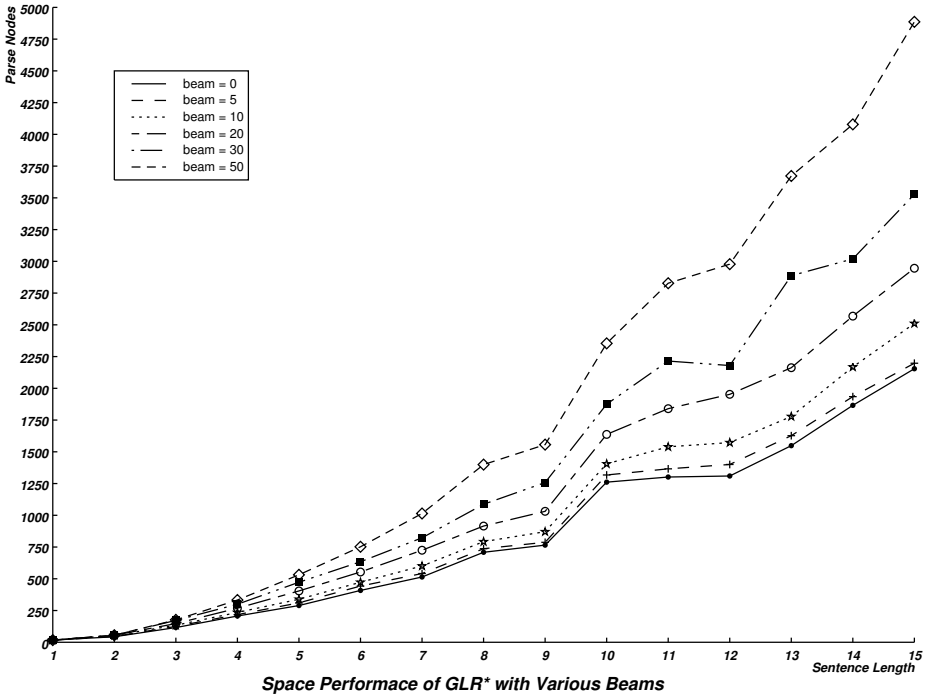


Figure 3.17 Space Performance of GLR* with Various Beam Widths as a Function of Sentence Length

Chapter 4

Statistical Disambiguation

4.1. Introduction

Wide coverage practical natural language grammars often produce large numbers of grammatical analyses for a given input. This is due partly to inherent structural ambiguities in natural language, and partly to the fact that it is difficult to integrate semantic interpretation directly into the parsing process, in a way that would limit the resulting analyses to only a small set of semantically plausible ones. Thus, many natural language understanding systems are designed in a way where semantic interpretation is performed by a separately designed module, which is applied on the set of grammatical analyses produced by the parser. Such a module can then select a particular parse analysis that translates into the most plausible interpretation, and may take advantage of additional sources of knowledge, such as the discourse.

The amount of post parser work required can be cut down considerably if the parser can be trained to identify and prune out grammatical analyses that are "unlikely" to produce a meaningful interpretation, or at least to rank the set of grammatical analyses according to such measures of likelihood. It is primarily for this reason that probabilistic parsing and statistical disambiguation methods have received much attention in recent years.

The abundance of parse ambiguities for a given input is greatly exacerbated in the case of the GLR* parser, due to the fact that the parser produces analyses that correspond to many different maximal (or close to maximal) parsable subsets of the original input. It is therefore imperative that the the parser be augmented with a statistical disambiguation module, which can assist the parser in selecting among the ambiguities of a particular parsable input subset. We will later see how the statistical disambiguation module can be useful in ranking analyses that correspond to *different* parsable subsets of the input (see Chapter 5).

This chapter describes a newly developed statistical disambiguation module, designed to augment the unification-based versions of both the GLR and GLR* parsing systems. After reviewing the major current approaches to probabilistic parsing and statistical disambiguation, we describe the statistical framework upon which our disambiguation module is based. Our statistical disambiguation method is similar in principle to a method proposed by Carroll [11], where probabilities are associated *directly* with the actions of the parser, as they are defined in the pre-compiled LR parsing table. At runtime, these probabilities are used to induce probabilities on the various analyses considered by the parser. We then describe the training procedures for collecting the necessary statistical data for a particular grammar. The probabilities of the various actions in the LR parsing table are estimated from a corpus of disambiguated parses. Next, we describe how the collected statistical data is used in order to disambiguate among the various analyses produced by the parser

at runtime. Finally, we present the results of an evaluation of the performance of the statistical disambiguation module on several different grammars.

4.2. Previous Work

4.2.1. Principle-based Methods

Several different principle-based disambiguation methods have been investigated in the literature. The most well known of these are Right Association and Minimal Attachment.

According to the principle of Right Association [44] [20] [67], it is preferable to attach a terminal symbol to the lowest right-most non-terminal node to which it can attach. This has the effect of grouping the terminal symbol with those immediately to its left. For example, in the sentence "*John said he saw Mary yesterday*", the preference is to attach the adverb "*yesterday*" to the verb "*saw*" and not to the the verb "*said*".

The principle of Minimum Attachment [44] [20] [67], on the other hand, prefers to attach a terminal symbol into a parse tree with the fewest possible number of new non-terminal nodes linking it with the nodes already in the tree. This disambiguation criterion is thus dependent on the way the particular grammar was written, and different grammars will disambiguate differently based on this principle. Nevertheless, with reasonable grammatical descriptions, Minimal Attachment has been shown to correctly disambiguate various types of ambiguities. Right Association and Minimal Attachment can at times be in conflict, each prefering a different ambiguity.

Right Association, Minimal Attachment and other principle-based disambiguation methods primarily attempt to explain the underlying rules used by humans in sentence comprehension and disambiguation. These methods can then be applied to perform similar disambiguation tasks in computational environments. Gibson [25] describes how these and other properties can be combined into a partially unified theory of human sentence comprehension.

4.2.2. Statistical Language Models

The surge in work on probabilistic parsing methods in the last decade can be attributed to a large extent to major technological advancements in the area of speech recognition. Statistical language models have proven to be effective tools in many speech recognition systems. Such speech systems usually consist of two major components. The first is an acoustic component that matches the acoustic input to word models in its vocabulary. It produces a set of the most plausible word candidates, each associated with a probability. The second component of a speech recognition system is a language model, which given a list of previously hypothesized words, determines the probability of a word occurring next. Accurate language models are essential to the performance of the speech system.

One particular acoustic modeling method that has received much attention in recent years is Hidden Markov models (HMMs). An HMM is a finite state automaton (FSA) augmented with a set of state transition probabilities, and a set of state output probabilities. Speech systems use HMMs to model the acoustic signals produced by the process of human speech. These models can then be used to probabilistically reconstruct word candidates given the acoustic data. The two most important algorithms required for this reconstruction are the Viterbi algorithm [100], and the

Baum-Welch algorithm [7]. The Viterbi algorithm provides an efficient solution to the problem of finding the most probable state transition sequence for a given input and HMM. The Baum-Welch algorithm is an iterative procedure for re-estimating the probabilities of an HMM, in order to best account for a corpus of training data. A good overview of HMMs, the Viterbi and Baum-Welch algorithms, and their applications in speech recognition can be found in Rabiner [80].

Traditionally, simple statistical models, known as *n-grams* have been used for the language model component of the speech system. With an n-gram model, the probability of a word occurring next is estimated based on the $n - 1$ previous words. The simplest of these models is the bigram model, where this probability depends solely on the preceding word. Although the probabilities of n-gram models can be acquired and calculated from large training texts, these models are limited in their accuracy. The number of parameters that need to be estimated for trigrams and higher n-grams is enormous and may greatly exceed the size of the available text corpora. This can result in an inadequate model due to sparse data. Various methods have been proposed to deal with this issue [42] [13] [22] [49], with some degree of success. However, n-grams model the language on a superficial level, that captures very little of the linguistic constraints typical of any natural language.

N-gram models have also been used for other natural language tasks. Church [13] developed a probabilistic program for automatically tagging the words of input sentences into their parts of speech. The program uses gathered statistical information on part of speech trigrams and a dynamic programming technique to determine the part-of-speech tags most probable for a given input sequence of words. Church also presents a probabilistic method for determining the boundaries of noun phrases with high accuracy, using a stochastic analog of precedence tables. Along the same lines, work by Brill et al [9] uses distributional analysis on part of speech n-grams to determine constituent boundaries.

Statistical language models such as n-grams may be viewed as a way in which distributional analysis serves as an alternative to unavailable more sophisticated and detailed knowledge (syntactic, semantic or pragmatic) about the language. Probabilistic parsers, therefore, offer a potential for a higher degree of accuracy, since the structure of the language is modeled directly by a grammar, while the distributional analysis can still take the place of unavailable semantic and pragmatic knowledge.

4.2.3. Parsing with Probabilistic Context-Free Grammars

A probabilistic context-free grammar (PCFG) is a CFG, where the frequency of occurrence of grammatical phenomena has been modeled by assigning probabilities to the rules of the grammar. If this process is done in a consistent way, it induces a probability distribution on the sentences of the language defined by the grammar. More importantly though, it induces a probability distribution on the various derivations of a particular grammatical sentence, and can thus be used as a disambiguation method.

Supervised and Unsupervised Training Methods

One possible way of estimating the rule probabilities of a PCFG is by using a corpus of disambiguated parses. The frequencies of occurrence of the rules in the *correct* (disambiguated) parses

can be determined from the corpus. These frequencies can then be normalized into rule probabilities. This method can produce accurate rule probability estimates when trained on a sufficiently large corpus of disambiguated parses. The major problem with this *supervised* training method is that the process of constructing a sufficiently large corpus of disambiguated parses can be very labor intensive.

As an alternative to the supervised training method, an *unsupervised* efficient version of the Baum-Welch algorithm has been developed for PCFGs [37] [54], where the grammar is restricted to being in Chomsky Normal Form (CNF), and using the CKY parsing algorithm [112]. This iterative training algorithm, also known as the Inside-Outside algorithm, works in the following way. Each sentence is parsed, and all possible analyses are produced. A probability is then computed for each analysis, using the rule probabilities from the previous iteration. Instead of counting the occurrences of the grammar rules in only the *correct* parse of each of the sentences, the frequency of occurrence of the grammar rules is estimated by counting their occurrence in *all* analyses, each weighed by the probability of the analysis. The estimated frequencies are then normalized into a new set of rule probabilities, and the entire process is then repeated.

After a bounded number of iterations, the Inside-Outside algorithm is guaranteed to converge to a local optimum that maximizes the probability of the training corpus. However, there is no guarantee that a global optimum will be found, and the initial probabilities chosen for the rules of the grammar have a crucial influence on the converged probabilities [54]. Although the resulting rule probability estimates can be skewed due to bad initial values and having been trained on incorrect as well as correct parse analyses, the Inside-Outside method has the great advantage of being unsupervised, in the sense that it does *not* require a disambiguated training corpus.

Recent Work on PCFGs

There have been several works on parsing with PCFGs in recent years. Fujisaki et al [21] describe an experiment in which a PCFG was used as a disambiguation method. The grammar contained 7550 rules in CNF. The rule probabilities were determined using the iterative Inside-Outside method on a corpus of 4206 sentences. The collected statistics where then used to disambiguate sentences with multiple analyses, by selecting the analysis with the highest probability. This is done using a version of the Viterbi algorithm [37], which can find the n most likely parses of a given input sentence. Evaluation was manually performed on a set of 84 sentences drawn from the original training set. In 72 out of the 84 sentences (85%), the most probable analysis was the correct one. Of the remaining 12 sentences, in 6 cases none of the analyses found by the parser was correct, while in the other 6 cases, the correct analysis was not the most probable one.

Corazza et al [16] discuss how to compute the probabilities of partial strings and substrings of sentences derived from a PCFG. They also describe how such probabilities can be used in conjunction with an island-driven probabilistic parser to score alternative acoustic hypotheses produced by a speech recognition system.

Schabes and Waters [84] introduce stochastic lexicalized CFGs (SLCFGs), a context-free version of stochastic lexicalized tree-adjoining grammars, and present algorithms for parsing, training of the probabilities and recovering the most probable parse of a given input for these kinds of grammars. The main advantage of using SLCFGs is their lexical sensitivity, which provides them with a better basis for capturing distributional information about words.

GLR Parsing with Probabilistic CFGs

Several works have looked into the issue of how to integrate the computation of probabilistic information into the process of parsing with a GLR parser. The GLR parsing strategy can be applied to a probabilistic context-free grammar if the parser generator can convert the rule probabilities into a form suitable for the action table and consistent with the way the probabilities of the rules multiply in a derivation. Wright and Wrigley [110] demonstrate how this can be done for the various types of LR table construction methods for grammars without left recursion.

The probabilities assigned to the actions in the parse table must ensure that the product of the probabilities of the parsing actions that correspond to a particular parse tree is identical to the probability that would be assigned to the parse tree directly by the PCFG. At run time, probabilities must be associated with the partial parses in the GSS data structure so that the final parse tree probabilities can be computed incrementally in a compositional way. Also, partial parses with probability below a certain threshold can be discarded. Wright, Wrigley and Sharman [109] deal with some of these technical issues. They also describe how to efficiently find the most probable analyses of a given input string. Some additional techniques of how to efficiently handle a probabilistic version of the GSS, and a method of dealing with left-recursive grammars were developed by Ng and Tomita [75].

Wright [108] describes how probabilistic input, such as that originating from a speech recognizer can be handled in the context of the probabilistic version of the GLR parser.

None of the above mentioned works describes any large scale empirical evaluation of using PCFGs in the context of GLR parsing, for either statistical disambiguation or parsing of input from a speech recognition system.

4.2.4. A Refined Probabilistic Model for GLR Parsing

Carroll [11] identifies several problems with PCFGs, and proposes a refined probabilistic model for GLR style parsers, that can better address some of these problems. The main problem with PCFGs is that the probabilities are modeled on the level of the context-free rules. Thus, the probabilities can reflect a general preference for expanding a non-terminal A using a rule $A \rightarrow \alpha$ versus some other rule $A \rightarrow \beta$. However, there is no way to probabilistically capture the dependency of such preferences on the *context* in which the rule is being applied. As an example, Carroll looks at rules that expand noun phrases (NPs). Although it has been shown that a noun phrase is more likely to be expanded into a pronoun when occurring in the subject position of a sentence rather than anywhere else, this fact cannot be captured by a PCFG, since the relevant rule has a single global probability assigned to it.

Carroll then makes the observation that LR parsers implicitly encode some partial contextual information in their states. Therefore, if probabilities are modeled directly at the level of LR states, certain contextual preferences will be reflected in the modeled probabilities. For example, in the case of the noun phrases, the parser is in one state when encountering the NP in the subject position, and in another state when it encounters an NP in the object position. Therefore, by having the probabilities depend on the state of the parser, the preference for expanding an NP into a pronoun when it occurs in the subject position can potentially be captured through the modeled probabilities of the various states.

Carroll proceeds to propose a refined probabilistic model for GLR parsers, where probabilities are associated *directly* with the actions of the parser, as they are defined in the pre-compiled parsing table. The parsing table is viewed as a non-deterministic finite state machine. Each parse tree that can be produced by the parser has a one-to-one correspondence to a particular sequence of transitions in the defined finite-state machine. Therefore, the probability of an analysis can be computed from the probabilities of the sequence of transitions performed in the course of obtaining the analysis.

Carroll proposes using a *supervised* method for training the probabilities of the LR table actions. For this purpose, a corpus of disambiguated parses must be collected. In Carroll's system, this task is performed using an interactive version of parser, which operates using the very same grammar for which probabilities are to be trained. The interactive parser requires the user to make choices at points where the parser has more than one possible alternative action. Using some sophisticated lookahead techniques, Carroll manages to resolve many of these choice points automatically and thus eliminate a fair amount of necessary interaction with the user.

The corpus of disambiguated parses is used to directly compute the probabilities of the various actions in the LR parsing table. Carroll then describes the probabilistic version of his GLR parser, and deals with the issues of the construction of a probabilistic parse forest, and the process of probabilistic unpacking to retrieve the analysis with highest overall probability. Since the statistical framework of our GLR parsing system is very similar to that of Carroll, these issues are covered in detail in the following section, and both the similarities and differences of the two approaches are made explicit.

The experimental evaluation of Carroll's system was conducted on a test corpus constructed from a subset of noun definitions from the Longman Dictionary of Contemporary English (LDOCE). The definition sentences are on average 10 words long. 150 definitions were used for training, for which a disambiguated parse corpus was constructed using the interactive GLR parser. The action probabilities were calculated from the disambiguated parses. The training corpus was then reparsed with the probabilistic GLR parser, and the probabilistically highest ranking analysis of each definition was compared with the correct analysis taken from the disambiguated corpus. 3 of the 150 sentences could not be parsed with probabilistic parser. Of the parsable 147 sentences, the highest ranked analysis matched the correct analysis in 99 cases, which amounts to a success rate of 67%. On sentences of length 10 or less, the success rate is 76%. On a separate test set of 55 additional definitions of length 10 or less, a similar success rate of 75% was observed.

4.3. The Statistical Framework

The statistical framework that we use in our statistical disambiguation module is very similar to the one developed by Carroll [11], that was described in the previous section. In this section we describe this framework in detail.

Similar to Carroll's approach, we attach probabilities directly to the alternative actions of each state in the pre-compiled LR table. Because the state of the LR parser partially reflects the left and right context of the parse being constructed, modeling the probabilities at this level has the potential of capturing contextual preferences that cannot be captured by PCFGs. For example, a reduce action by a certain grammar rule $A \rightarrow \alpha$ that appears in more than one state can be assigned a different probability in each of the occurrences.

However, as Carroll notes, assigning a single probability to each reduce action per state does not make maximal use of the context encoded in the LR table. To a great extent, the context in which the reduction occurs is reflected via the *goto state*, the state reached after the reduction has been applied. The goto state is deterministically defined by the state recovered from the stack after popping the right-hand side of the rule, and by the rule's left-hand side non-terminal symbol. The goto state can distinguish, for example, whether a rule such as $NP \rightarrow pronoun$ occurs in the subject or object position of a sentence. It is therefore highly advantageous to refine the probabilities attached to reduce actions, and subdivide them according to the goto state. This is particularly true of the LR(0) tables used in the GLR and GLR* parsers, since reduce actions in LR(0) tables are not divided according to any lookahead.

Table 4.1 shows a probabilistic version of the parsing table constructed for the grammar in Figure 3.2. Due to the fact that this is a "toy" grammar, the probabilities in this table were not collected from an actual corpus and are shown merely for demonstration purposes. Note that the probability of a reduce action is divided among the possible goto states following the reduction. The "Goto" part of the parsing table is deterministic, and is therefore not displayed.

The refined approach just described corresponds to associating probabilities with the *transitions* of the finite-state automaton defined by the LR table, instead of just the actions defined in the action part of the LR table. This leads to a probabilistic model that views the LR table as a probabilistic finite-state machine.

In a finite-state probabilistic model, the probability of a particular transition is dependent *solely* upon the current state. Therefore, for each state, the sum of the probabilities of all possible transitions must add up to one. Because the probability of a transition depends only on the current state, the probability of a *sequence* of actions of the finite-state machine is simply the product of the individual action probabilities. This defines a probability space on the set of all possible transition sequences that originate from the start state of the machine. Because the graph of transitions of the finite-state machine can contain cycles[1], this set of transition sequences is likely to be infinite. In the resulting probability space, the sum of the probabilities of all possible transition sequences adds up to one.

It is important to note that modeling the parser as a probabilistic finite-state machine is an over-simplification. In reality, parser actions are dependent on more than just the current state. The current stack content, as well as the next input word restrict the set of possible transitions from the current state to a subset of the entire set of possible transitions. However, this model appears to be detailed enough to capture many of the structural preferences of the language, yet simple enough to enable reasonably adequate training of the probabilities from available corpus data.

Parse Disambiguation Using the Finite-State Model

We have yet to describe how the above finite-state probabilistic model can be used to disambiguate parses. Due to the way GLR parsers operate, there is a one-to-one correspondence between every parse tree that can be produced by the parser and a particular sequence of transitions in the defined finite-state machine. This allows us to associate the probability of the transition sequence with it's corresponding parse tree. Alternative analyses of a particular grammatical input sentence are

[1]Note that the LR table constructed for a grammar that generates an infinite language will necessarily correspond to a finite-state machine with cycles.

	Reduce	Shift				
State		det	n	v	p	$
0		sh3 (.87)	sh4 (.13)			
1						acc (1.0)
2				sh7 (.68)	sh8 (.32)	
3			sh9 (1.0)			
4	r3 (2 .36)					
	r3 (10 .53)					
	r3 (11 .11)					
5	r1 (1 1.0)					
6	r4 (2 .18)					
	r4 (10 .45)					
	r4 (11 .37)					
7		sh3 (.77)	sh4 (.23)			
8		sh3 (.62)	sh4 (.38)			
9	r2 (2 .40)					
	r2 (10 .32)					
	r2 (11 .28)					
10	r5 (5 .74)				sh8 (.26)	
11	r6 (6 .69)				sh8 (.31)	

Table 4.1 Probabilistic Parsing Table for Grammar in Figure 3.1

manifested via a set of possible parse trees. Each possible parse tree will correspond to a different sequence of transitions, from which it can inherit a probability. These probabilities can then be used for disambiguation.

Because we use a finite-state probabilistic model, the action probabilities do not depend on the given input sentence. The relative probabilities of the input words at various states are implicitly represented by the action probabilities. For example, a certain state may allow several different shift actions, one for each possible next occurring input word. The probabilities attached to these shift actions reflect the relative frequency of occurrence of the various possible words, as well as preferences with respect to other possible actions from the state, such as reduce actions. Because the finite-state probabilistic model assigns an actual probability to every transition sequence, transition sequences that correspond to parses of *different* inputs are directly comparable. This enables us to use the statistical parse scores as one of a set of evaluation measures for selecting among the parses of various parsable subsets of a given input, which are produced by the GLR* parser. This topic is discussed in detail in the next chapter.

One should note that transition sequences that correspond to parse trees have unique properties[2], and thus constitute only a subset of the set of *all* transition sequences. Due to this fact, the finite-state probabilistic model does not define a probability space over the set of all possible parse trees of grammatical sentences. It therefore does not constitute a *language model* - a probabilistic model

[2]For example, they must end in the final accepting state.

for the language defined by the grammar. In particular, the probability of a grammatical sentence cannot be directly computed as a probability sum over all possible transition sequences that lead to valid parses of the sentence.

Even more importantly, the statistical score attached to a particular parse of a sentence cannot be considered a true parse probability, since the scores over all alternative parse trees of a grammatical sentence do not sum to one. This problem could be corrected by normalizing the scores of alternative parses of a sentence. However, as noted by Carroll [11] (pp. 122), the task of parse disambiguation does not necessitate a true language model. It is sufficient that the statistical information being used reflect the underlying structural grammatical preferences. The disambiguation procedure can then select among competing analyses, using the statistical scores as indicators of these preferences.

The success of our disambiguation method therefore depends on the degree to which the finite-state probabilistic model has the following property:

> **Property 2** *The Reflection Property:* Statistical preferences between alternative parse trees of ambiguous grammatical sentences can be captured by a probabilistic finite-state model, in a way that allows the probabilities of the transition sequences that correspond to the alternative parse trees to reflect the original grammatical preferences.

The fact that disambiguation procedures that are not based on valid language models can in fact yield more effective results has been observed by other researchers as well. For example, *Pearl* [61] and *Picky* [62] use disambiguation procedures in which parses are scored based on the geometric mean of conditional probabilities of the rules that appear in the parse derivations. Context is captured by conditioning the rule probabilities on the left and right neighboring non-terminals in the dominating rule, as well as on the part-of-speech trigram at that point. By using the geometric mean of the rule probabilities instead of their product, the researchers claim that their disambiguation procedure can better handle the effects of dependency between constituents. Also, the predisposition towards parses with shorter derivations is neutralized.

4.4. Training of the Probabilities

4.4.1. Supervised versus Unsupervised Training

We now turn to deal with the issue of how to obtain good and accurate probabilistic information from training data. Similar to the case of PCFGs, a decision must be made whether to use a *supervised* or an *unsupervised* training method. This decision is directly related to the question of whether our probabilistic model must be trained on *correct* data obtained from a corpus of disambiguated parses, or whether training on all possible analyses of a corpus of input sentences can also yield meaningful statistics.

In principle, a Baum-Welch style unsupervised training procedure could be developed for the probabilistic finite-state model. However, it is very unlikely that an iterative training procedure on all possible analyses of a corpus of input sentences would result in useful probabilistic information. Baum-Welch iterative re-estimation maximizes the probabilities of the sentences in the training corpus for a given grammar. In the case of our finite-state model, the training would maximize the sum of the probabilities of all transition sequences that correspond to analyses of sentences in the training corpus. However, since the modeled probabilities are to be used to select the *correct* parse

tree among the alternative analyses of a given input, our goal should be to maximize the probability of the correct analysis, given a sentence in the training corpus. Baum-Welch re-estimation is not likely to achieve this goal.

For example, assume a particular structural ambiguity is manifested in the parsing table through a choice between a shift action and a reduce action from a particular state. Let us assume that taking the shift action results in a correct analysis, while taking the reduce action results in an incorrect parse tree. Training on *correct* parse trees of sentences in which this ambiguity occurs should result in an overwhelming probability bias towards the shift action, since this action will appear in all transition sequences that correspond to the correct parse trees of the ambiguous sentences. However, if the a-priori probability of the shift and the reduce actions are the same, these probabilities will have an equal effect on the probabilities of both the correct and incorrect transition sequences of the ambiguous sentences. Although small initial biases, or other effects, may tilt the re-estimated probabilities of the two actions in one way or another, there is nothing in the Baum-Welch training method that directly encourages the creation of a probability bias towards the correct action.

It is therefore quite evident that only a supervised training method, where the statistical information is collected from transition sequences that correspond to *correct* analyses, is likely to result in probabilistic information that can reflect structural grammatical preferences. We have therefore developed such supervised training methods for our statistical model.

4.4.2. Obtaining a Disambiguated Training Corpus

The quality and usefulness of the statistical information in our probabilistic model depends to a great extent on being able to train the probabilities from an adequately large and accurate corpus of disambiguated correct parses. However, obtaining such a corpus is a tedious labor intensive process. As other researchers have noted [11] [58], the manual construction from scratch of a large scale database of parse trees can be an infeasible task. This is even more so in systems in which the grammar is being incrementally developed and the actual "correct" parse trees of sentences will be constantly changing along side with the grammar modifications. In this case, in order to train the probabilistic model for a new version of the grammar, a new corpus of correct parse trees would have to be constructed.

We have managed to develop some techniques that significantly alleviate these problems in the case of the unification-based GLR parsing system. As previously described in Chapter 2, the parsing system supports grammatical specification in an LFG framework, that consists of context-free grammar rules augmented with feature bundles that are associated with the non-terminals of the rules. For each context-free rule, the grammar formalism allows the specification of the feature structure associated with a left-hand side non-terminal of the rule as a function of the feature structures that are associated with the non-terminals on the right-hand side of the rule. The feature structures that are computed at parse-time are attached to their corresponding parse node structures. At the end of parsing an input sentence the parser returns a list of the top parse nodes in the parse forest, as well as the computed feature structure associated with each of these nodes.

The feature structure associated with a node at the root of the parse forest is in fact the most important part of the parse result. In all of the applications that have been developed using the GLR parsing system, the grammars were designed to produce output feature structures that contain all structural information from the parse that is required by the application.

The correctness of a parse result in our parsing system is ultimately determined according to the feature structure and not the parse tree. Therefore, we are primarily interested in parse disambiguation at the feature structure level. Most structural ambiguities in the grammar correspond to ambiguities at the feature structure level as well. However, structural ambiguities that do not produce an ambiguous feature structure are of little interest. It is therefore more appropriate in our system to collect a corpus of sentences and their correct feature structures, as opposed to sentences and their corresponding parse trees.

Maintaining a collection of correct feature structures has several additional advantages. In many cases, the specification of correct feature structure output is designed in advance, and is independent of the actual grammar development process. As a result, feature structure output is likely to be rather stable, and the correct feature structures for the corpus of training sentences is not likely to change much in the course of grammar development. Thus, regardless of whether the corpus of correct feature structures is initially constructed manually from scratch, or semi-automatically from the parse results of an early version of the grammar, once it exists, it should be fairly easy to maintain throughout the process of grammar development.

Our primary investigated application in this thesis is the JANUS speech-to-speech translation system (see Chapter 6). In JANUS, a large collection of sentences and their correct feature structures was constructed for development and evaluation purposes, and was thus directly available for training the probabilities of the statistical model. For other applications such as the ATIS domain, we developed an interactive disambiguation procedure for efficiently constructing a corpus of disambiguated feature structures.

Our experience has shown that choosing the correct feature structure is also much easier than selecting a correct parse tree. This can significantly reduce the amount of manual work required for building a collection of disambiguated parses when using the interactive disambiguation procedure.

The Interactive Disambiguation Procedure

To assist a user in building a corpus of correct feature structures, we developed an interactive disambiguation procedure. In the context of this procedure, the GLR parser is used in a mode in which both skipping and ambiguity packing are disabled. Skipping is disabled because we anticipate the disambiguation corpus to consist of only grammatical sentences, which are parsable in full. This allows the output set to consist of only the ambiguous analyses of the full input sentence. Ambiguity packing is also disabled, so that ambiguities will not be embedded within the feature structure, and the complete set of alternative analyses will be available at the top level.

The disambiguation procedure operates on a set of input sentences by processing them one by one. Each input sentence is parsed by the parser. The set of all alternative feature structure analyses is then presented to the user. If the parse was not ambiguous, the user needs only to verify that the computed feature structure is indeed correct. If more than a single analysis was found, the user may choose which, if any, of the set of analyses found is the correct one. If a correct feature structure was selected, the system stores it, along with the original input sentence. If none of the parsed analyses was correct, no information is stored.

Our interactive disambiguation procedure computes the complete set of analyses for each parsed sentence. This method has two potential drawbacks. First, the set of all analyses for some sentences may be very large, burdening the user with a tedious selection process. Second, producing the complete set of analyses can be computationally expensive, particularly since no ambiguity packing

is performed. In the applications that we have experimented with in the course of this thesis work, neither of these two issues proved to be a serious problem.

While building a collection of disambiguated parses, the interactive disambiguation procedure simultaneously collects statistical information required for calculating the transition probabilities for the finite-state probabilistic model. When the user selects the correct feature structure of a parsed sentence, a secondary procedure is called, that converts the selected feature structure into the corresponding correct transition sequence. The method by which this is done is described in the next subsection. Once the correct sequence of transitions is known, frequency counters for the transitions can be updated. Statistics for the probabilistic model can then be incrementally calculated on the partial corpus collected so far.

Using statistics collected on a partial training corpus while running the interactive disambiguation procedure can further ease the process of selecting the correct feature structure. The existing statistics are used to score and rank the set of alternative feature structures produced by the parser. These are then presented to the user ordered by their scores. It is likely that the correct feature structure will be found among one of the top ranking analyses, shortening the required amount of browsing. Since the quality of the statistics should fairly monotonically improve with the size of the training corpus, it is worthwhile to every so often recalculate updated statistics from the partial corpus collected so far, and use the updated statistics for further enlarging the training corpus of disambiguated feature structures. The training corpus can thus be incrementally built over a prolonged period of time, while statistics derived from the existing training corpus can already be utilized for disambiguation.

Carroll [11] suggests an alternative method of interacting with the parser in order to find the correct parse. According to this method, while parsing the sentence, the user is asked to make a choice whenever the parser encounters a point of non-determinism with multiple alternative actions. Using some sophisticated lookahead techniques, Carroll manages to resolve many of these choice points automatically and thus eliminate a fair amount of necessary interaction with the user. This approach has the disadvantage of requiring the user to resolve local ambiguities that might not appear in the final set of analyses. Furthermore, the local information available at each choice point may not be sufficient to allow the user to easily make the correct choice.

These two disadvantages are particularly serious in our GLR parsing system. Due to the fact that our system uses LR(0) parsing tables, our parsing system normally attempts a very large number of reduce actions in the course of parsing a sentence. In many cases, a majority of these reductions fail due to feature structure unification failure. As a result, the amount of interaction with the user that would be required for on the fly action disambiguation would be far greater than that required for post selection. We therefore chose not to pursue an approach similar to that of Carroll for our interactive disambiguation procedure.

4.4.3. Deriving Probabilities from a Training Corpus

Once we have a corpus of input sentences and their disambiguated feature structures, we can train the finite-state probabilistic model that corresponds to the LR parsing table. The process by which this is done first involves accumulating counters that keep track of how many times each particular transition of the LR finite-state machine was taken in the course of correctly parsing all the sentences in the training corpus. After processing the entire training corpus and obtaining the counter totals, the counter values are treated as action frequencies and converted into probabilities

by a process of normalization. A smoothing technique is applied to compensate for transitions that did not occur in the training data. We now describe the details of this process.

Finding the Correct Transition Sequence

In order to be able to accumulate counters for the LR table transitions, a method must be developed by which a correct feature structure of a parsed sentence can be converted back into the sequence of transitions the parser took in the process of creating the correct parse. This is done in the following way.

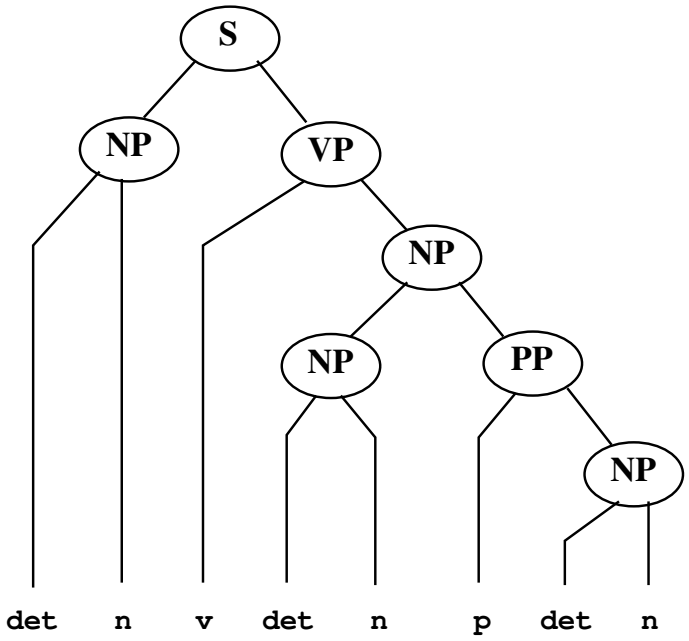
1. The correct parse tree that corresponds to the correct feature structure is recovered.
2. The correct transition sequence is determined from the correct parse tree.
3. Frequency counters for all transitions that appear in the correct transition sequence are updated.

In order to recover the correct parse tree, the input sentence is once again parsed, with the parser set in the mode in which ambiguity packing is disabled. This results in a parse forest which consists of a set of trees, each corresponding to an alternative analysis of the sentence. A matching procedure is then used to compare the feature structure of each of the parse trees with the correct feature structure that was stored in the training corpus. When a matching feature structure is found, it's corresponding parse tree is retrieved from the parse forest.

Once the correct parse tree has been found, it is converted into an action sequence that corresponds to the rightmost derivation in reverse performed at runtime by the parser. This is done by a postfix procedure that traverses the parse tree. Each leaf node in the tree corresponds to a shift action, and each final visit to an internal node corresponds to a reduce. The precise rule being reduced can be determined from the non-terminal at the node, and the node's direct children. For example, consider the parse tree in Figure 4.1, constructed for the sentence "`det n v det n p det n`", according to our simple example grammar from Figure 3.2. The action sequence corresponding to the parse tree can be seen below the parse tree. We denote the shift actions by "`sh`", while reduce actions are denoted by "`r`i", where i is the rule number.

The complete transition sequence of the LR finite-state machine can then be reconstructed via a simulation of the parsing of the input by the parser. In this simulation, the correct actions of the parser are deterministically determined from the recovered action sequence. All that remains is to use the parsing table itself, starting from the start state, in order to determine the sequence of states through which the parser moves. The goto states that are part of the reduce actions are determined from the table using the simulated parser stack. The reconstructed transition sequence that corresponds to our example parse tree can be seen at the bottom of Figure 4.1. It was constructed by a simulation of the parser using the parsing table from Table 3.1. The actions are now interleaved between the parser states. A state that follows a reduce action is the determined goto state. The transition sequence starts in state 0, the start state, and ends in state 1, the final accepting state, where the last action is to accept the input.

Once the transition sequence of the LR finite-state machine has been recovered, counters for each of the observed transitions are incremented. A counter is maintained for every observed transition from each state. Shift actions are identified according to the token being shifted and the



Action sequence:

```
sh sh r2 sh sh sh r2 sh sh sh r2 r6 r4 r5 r1
```

Transition Sequence:

```
0 sh3 3 sh9 9 r2 2 sh7 7 sh3 3 sh9 9 r2 10 sh8
8 sh3 3 sh9 9 r2 11 r6 6 r4 10 r5 5 r1 1 acc
```

Figure 4.1 A Parse Tree with Corresponding Action and Transition Sequences

shift state. Reduce actions are identified according to the number of the rule being reduced and the resulting goto state.

Converting Transition Frequencies into Probabilities

When the process of collecting the frequencies of all observed transition sequences is complete, the frequencies can be converted into actual probabilities for the finite-state probabilistic model. For each state, the frequencies of all the transitions that occur from the state are converted into probabilities, so that their sum will add up to one. This is done by normalization - the frequency counts of all transitions from the state are added up to a total count, and the fraction of the total for each of the transitions is calculated.

Smoothing of the Probabilities

Even when training is done on a very large training corpus, there are likely to be many transitions of the finite-state model that will not have been observed even once. This is particularly true when working with large scale grammars in practical applications. The number of different transitions, for which frequencies are to be collected, can easily reach an order of magnitude of 10,000 or even 100,000 or more.

Without any smoothing of the probabilities, a transition that was not observed even a single time in the training corpus is effectively assigned a probability of zero. This has some undesirable consequences on the utility of the probabilistic model when used for disambiguation. In our investigated applications, it is quite common for the system to encounter an input sentence that is different in structure than all sentences that occurred in the training corpus. If in the course of parsing such a sentence, the parser performs a transition that was never observed during training, the parse will necessarily be assigned a probability of zero, since the probability of the entire transition sequence is the product of the individual transition probabilities, one of which is zero. If the parsed sentence is ambiguous, and all of the transition sequences that correspond to the ambiguities contain a zero probability transition, then all ambiguities will be assigned a probability of zero. In such cases, our statistical model will be useless for disambiguation.

As it turns out, this is a frequently occuring problem when using the GLR* parser. The very nature of the applications to which we apply GLR* is such that the input is likely to be noisy and ungrammatical. As a result, the parsable grammatical subsets that are processed by GLR* frequently create parses that did not appear in the training corpus. All such parses would necessarily be assigned a zero probability by our probabilistic model. In a JANUS evaluation of English analysis, conducted on an test set of unseen transcribed spontaneous speech, over 50% of the input sentences were assigned a probability of zero as a result of this problem. Thus, smoothing the probabilities to account for unseen transitions is extremely important.

Similar to Carroll [11], we use a simple smoothing technique, which is a variant of the Good-Turing smoothing method [27]. Unobserved transitions are given a frequency of one. The frequency of observed transitions is not modified. The probabilities of the various transitions is then computed by normalization. This has the effect of assigning to an unobserved transition a probability that is the reciprocal of the result of adding the number of unobserved transitions to the total number of observed transitions from the state. More sophisticated smoothing techniques that have been proposed for n-gram language models [42] [14] cannot be applied in our case due to the lack of an appropriate back-off model and the extreme sparseness in the data.

Even so, our smoothing technique is sufficient to correct the problem of zero probabilities in a significant number of cases. Particularly, it allows us to compute comparative probabilistic scores for the analyses of ambiguous parses in cases where all the ambiguities would have previously received a zero probability.

Figure 4.2 shows a small portion of the smoothed probability table for the English analysis grammar of the JANUS project. For practical reasons, the probability table is kept separate from the actual parsing table. The probabilities were trained from an actual corpus of disambiguated feature structures (see the evaluation section of this chapter for details). Shift transitions are identified by a pair consisting of the symbol "`S`" and the shift state. The pair is preceded by the token being shifted. The "wild-card" token is represented by the symbol "`%`". Reduce transitions are represented by a double pair, the first of which is the word "`REDUCE`" followed by the number

```
((0 (((% (S . 274)) 0.2958876629889669)
     ((I (S . 657)) 0.08926780341023069)
     ((WELL (S . 678)) 0.043129388164493 48)
     ((HOW (S . 149)) 0.038114343029087 26)
     ((LET (S . 537)) 0.033099297893681046)
     ((NO (S . 655)) 0.026078234704112337)
     ((THAT (S . 116)) 0.02106318956870612)
     ((*) 0.00100300902708124 38))
 (1 (((*) 1.0)))
 (2 ((((REDUCE 1) (<START> . 1)) 0.9986244841815681)
     ((*) 0.001375515818431912)))
 (3 ((((REDUCE 2) (<SENT> . 2)) 0.9986244841815681)
     ((*) 0.001375515818431912)))
 (4 ((((REDUCE 3) (<SENT3> . 3)) 0.9458204334365326)
     ((TOO (S . 688)) 0.006191950464396285)
     ((THOUGH (S . 690)) 0.0015479876160990713)
     ((THEN (S . 691)) 0.0015479876160990713))
     ((*) 0.0015479876160990713))
 (5 (((% (S . 274)) 0.06132075471698113)
     ((I (S . 657)) 0.0589622641509434)
     ((WE (S . 659)) 0.025943396226415096)
     ((THE (S . 93)) 0.01650943396226415)
     ((LET (S . 537)) 0.014150943396226416)
     ((THAT (S . 116)) 0.014150943396226416)
     ((*) 0.0023584905660377358))
 (6  ((((REDUCE 13) (<SENT1> . 173)) 0.9974489795918368)
     ((*) 0.002551020408163265)))
 (7 ((((REDUCE 14) (<SENT1> . 173)) 0.8)
     ((*) 0.2)))
 (8 ((((REDUCE 15) (<SENT1> . 173)) 0.6666666666666666)
     ((*) 0.3333333333333333)))
 (9 ((((REDUCE 16) (<SENT1> . 173)) 0.9333333333333334)
     ((*) 0.06666666666666667)))
 (10 ((((REDUCE 17) (<SENT1> . 173)) 0.9444444444444444)
      ((*) 0.05555555555555555)))
 (11 ((((REDUCE 18) (<SENT1> . 173)) 0.9)
      ((*) 0.1)))
 (12 ((((REDUCE 19) (<SENT1> . 173)) 0.875)
      ((*) 0.125)))
 (13 ((((REDUCE 126) (<S> . 6)) 0.9923857868020305)
      (((REDUCE 126) (<S> . 1503)) 0.005076142131979695)
      ((*) 0.0025380710659898476)))
```

Figure 4.2 A Portion of a Smoothed State Transition Probability Table

of the rule being reduced. The second pair consists of the rule's left-hand side non-terminal and the goto state. Unobserved transitions are represented by an asterisk. Since the probability of all unobserved transitions from a given state is the same, it need only be represented once for each state.

4.5. The Disambiguation Procedure

We now turn our attention to the question of how to use the probabilistic finite-state model for which we trained the statistics in order to score the alternative parse analyses that are produced by the parser at runtime. Once computed, these scores can be used for disambiguation, by unpacking the analysis that received the best probabilistic score from the parse forest.

4.5.1. Scoring Alternative Parse Ambiguities

As explained in the previous section, each parse tree of a given input sentence has a one-to-one correspondence with a sequence of transitions of the finite-state machine defined by the parser's LR parsing table. Since our finite-state probabilistic model defines a probability for every transition sequence, this probability can be associated directly with the corresponding parse tree. This scheme can then be used for disambiguation by selecting among competing analyses according to the probabilities attached to each of the corresponding parse trees.

Due to the fact that the probabilities we use do not form a probabilistic language model, the scores of all alternative analyses of a given input do not sum to one. This could be overcome by a process of score normalization. However, because we wish to use the probabilistic parse scores as a measure of comparison between analyses of *different* parsable subsets of the input, we do not normalize the scores. As noted earlier in this chapter, it is not inherently necessary to use a true language model for statistical disambiguation.

When Should Probabilistic Information Be Used?

Another decision that needs to be considered is whether probabilistic information is to be applied in advance of the parsing process, during parsing itself, or only as a post process, after the parser has completed constructing the parse forest. Applying probabilistic information at parse time has the potential for increasing the parser's efficiency by allowing the parser to prune out sub-parses that receive low probabilities. In advance of the parsing process, probabilistic information may also be used to prune out from the parsing table transitions that received very low probabilities. This has the potential of making the parsing table more deterministic and thus more efficient.

However, both of these possibilities turn out to be infeasible in the setting of the unification based GLR parsing system. To a major extent this is due to the non local effects of feature structure unification. At any point in the parse derivation process, a sub-analysis with a high probabilistic score may be discarded due to a unification failure. This has the effect of assigning the sub-analysis a probability of zero at the point of feature structure unification failure. As a result, a competing sub-analysis that previously was probabilistically sub-optimal may become optimal at this point. Due to these non local effects of unification, no Viterbi style optimization process can be applied to the parser at runtime.

Pruning out parser transitions in advance is infeasible for similar reasons. Additionally, the obvious candidates to be pruned out in advance would be transitions that were not observed in the training corpus. However, as noted earlier, parsing new unseen data will frequently involve some transitions that were not observed in the course of training. This is particularly the case in the context of the word skipping performed by the GLR* parser. If such transitions were to be pruned out, the parser would be prohibited from parsing significant portions of the data. We thus refrain from any advance pruning of the parsing table.

Since it is infeasible to use the probabilistic information in advance or in the course of the processing process, statistical disambiguation for the unification-based GLR parsing system is performed as a post-process, after the entire parse forest has been constructed by the parser.

Disambiguation as a Post-process

Computing the statistical scores of all alternative analyses is complicated due to the fact that they are represented in a *packed* forest. In the absence of ambiguity packing, a procedure similar to the one used for training could be used. Each alternative analysis could be scored in the following way. First, the parse tree corresponding to the analysis would be fetched from the parse forest. Second, our parse simulation procedure would convert the parse tree back to the sequence of transitions performed by the parser. Third, the probability of the transition sequence would be calculated and attached to the analysis. After each alternative analysis were scored in this fashion, the best scoring analysis could be selected.

However, due to ambiguity packing, the set of alternative analyses and their corresponding parse trees are *packed* in the parse forest. One possibility would therefore be to first unpack all alternative analyses from the parse forest, and then use the method outlined above to probabilistically score each of the alternatives, in order to select the best one. This approach is computationally infeasible. In order to avoid unpacking the entire forest, we must thus develop a method for computing statistical scores for the packed nodes of the parse forest directly, in a way that will allow us to unpack only the best scoring analysis in the forest.

4.5.2. Computing a Probabilistic Packed Parse Forest

In order to be able to assign probabilistic scores directly to the nodes of the packed parse forest, the following two problems must be addressed.

1. *How are packed nodes to be scored?*

2. *How can a node in the parse forest be correlated with the parser transitions that were carried out when the node was created?*

The scoring procedure that we have developed addresses the first question in the following way. For a packed node, a score is computed for each of the alternative sub-analyses that are packed into the node. The packed node itself is then assigned the "best" of these scores, by taking a maximum. This score will then be used to compute the scores of any parent nodes in the forest. According to this scheme, the score that is attached to the root node of the forest should reflect the score of the best analysis that is packed in it.

Finding "From" and "Shift/Goto" States

Correlating a node in the parse forest with the parser transitions that were carried out when the node was created is more complicated. In order to determine the transitions, we must find the state (or set of states) of the parser when the node was created, as well as the action that was taken by the parser. This is done using information available in the GSS. Symbol nodes in the GSS contain pointers to their corresponding parse nodes. By tracing the state nodes to which a symbol node is connected in the GSS, it is possible to determine the state (or states) of the parser and the action performed, when the symbol node was created. These states can then be used to determine the transition performed by the parser when the symbol node and its corresponding parse node were created.

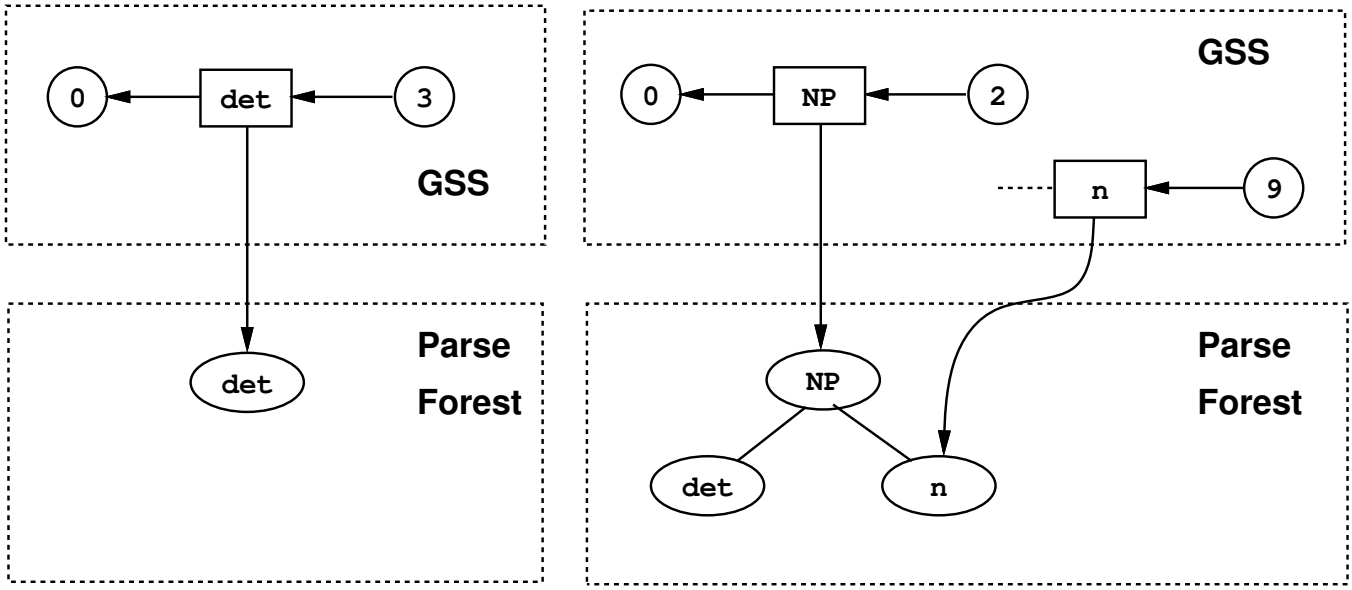
Figure 4.3 demonstrates how this is done. State nodes in the GSS are denoted by circles; symbol nodes in the GSS are denoted by squares; and parse nodes of the parse forest are denoted by ovals. For each parse node we determine a "from" state and a "Shift/Goto" state. The "from" state indicates the state of the parser *prior* to taking the action that creates the parse node. The "Shift/Goto" state indicates the state of the parser *after* the action that creates the parse node is performed. "Shift/Goto" states of parse nodes are determined by following pointers from state nodes of the GSS to symbol nodes and from there to the corresponding parse node. This can be seen in both parts of the figure. In part (a), the state node in the GSS with state 3 leads to the symbol node "det" which points to the parse node "det" in the parse forest. Therefore, the "Shift/Goto" state for the parse node "det" is 3. In part (b), the GSS state node with state 2 leads to the symbol node "NP" which points to the parse node "NP" in the parse forest. Thus, the "Shift/Goto" state for the parse node "NP" is 2.

Determining the "from" state is dependent on whether the action associated with the parse node was a shift or a reduce. If the parse node has no children, it corresponds to a shift action. This is shown in part (a) on the left of the figure. For shift actions the "from" state is determined from the corresponding symbol node in the GSS. The symbol node contains a pointer to the state node from which the symbol was shifted. In our example, this was the state node with state 0. Therefore, the "from" state for the parse node "det" is 0.

If the parse node has children, it corresponds to a reduce action. The precise rule being reduced can be determined from the parse node and its children nodes[3]. In the case of a reduction, the "from" state, is the "Shift/Goto" state of the right-most child of the parse node. In the example in part (b) on the right of the figure, the right-most child is the parse node "n", and its corresponding "Shift/Goto" state is 9. Thus, this becomes the "from" state of the NP parse node.

The "Shift/Goto" and "from" states of all parse nodes are determined by a procedure that works in two passes. In the first pass, the procedure runs through all state nodes in the GSS and for each of them determines the "Shift/Goto" states of the appropriate parse nodes. In the second pass, the procedure runs through all symbol nodes in the GSS and for each determines the "from" state of the corresponding parse nodes.

[3]The category of the parse node is the left-hand side non-terminal of the rule, while the categories of the children form the right-hand side of the rule.



(a) A Shift Action:
From state = 0
Shift/Goto state = 3

(b) A Reduce Action:
From state = 9
Shift/Goto state = 2

Figure 4.3 Determining "From" and "Shift/Goto" States for a Parse Node

Scoring the Parse Nodes

Once the "from" and "Shift/Goto" states of all parse nodes have been determined, the nodes can be scored. Using the "from and "Shift/Goto" state information and the actual shift or reduce action, the precise transition done by the parser is determined, and its probability is fetched from the smoothed probability table. If the action was a shift, the transition probability is assigned to the parse node as a score. If the the parse node has children, the corresponding action was a reduce. In this case, the probability of the reduce transition is multiplied with the scores of the children nodes to compute the score of the corresponding subtree, which is then attached to the parse node.

As already mentioned, the score of a packed node is determined by calculating the score of each of the alternative packed sub-analyses, and then selecting the score of the maximal scoring one.

Because the score computation of an internal node in the parse forest requires the prior computation of the scores of the node's children, parse node scoring is performed in a "bottom-up" fashion. Since parse nodes are sequentially allocated at parse time from a dynamic array, scoring the parse nodes in the order in which they were created can be done by processing the array from beginning to end.

Over-estimated Parse Node Scores

The scoring method that we have just outlined should, in principle, assign to each node in the parse forest a score that reflects the probability of the most probable parse subtree rooted at the node.

However, the score may in some cases be an *over-estimate* of the probability of the actual best subtree, due to two reasons.

The first reason is that the scoring process does not take into account the effects of feature structure unification. As a result, the score attached to a parse node could in fact correspond to a potentially valid sub-analysis according to the phrase structure rules, that was discarded at some point due to feature structure unification failure. When the failure of feature structure unification causes a sub-analysis to be discarded, it in effect rules out the particular parse nodes that composed it. Since our parse forest ambiguities are packed, discarding a sub-analysis is equivalent to ruling out the corresponding ambiguities in the packed nodes of the forest. This may effect the score of a packed node, if the score was determined by the ambiguity that is being ruled out. In such cases, the scores of all ancestor nodes of the affected packed node are also affected. However, because it would be computationally expensive to do so, we do not take these non-local effects of unification failure into consideration when calculating the parse nodes scores.

The second reason has to do with the "from" states that are used to determine the precise parser transition taken when the parse node was created. Due to the efficient representation of the GSS, symbol nodes are at times connected to more than a single incoming state node. In such cases, the parse node corresponding to the symbol node will have a set of "from" states instead of just a single unique state. Such situations occur when the parse node is in fact shared among several different analyses, with a different "from" state corresponding to each. However, we wish to assign the parse node a single score, regardless of the analyses into which it is incorporated. We therefore use a maximization technique. Instead of a single parser transition, we determine a set of possible transitions using the set of "from" states. We then select the transition of maximal probability and use this probability in calculating the score that is assigned to to the parse node. Consequently, this may have the effect of assigning an overly optimistic probability to the parse node.

Because our scoring method is imprecise in the ways just described, the score assigned to a parse node it is not guaranteed to be the actual true probability of the best sub-analysis rooted at the node. However, the score is always a close over-estimate of this probability. It may thus be viewed as a heuristic for determining the actual best scoring ambiguity packed in the forest. Our experiments have shown that using this heuristic for picking the ambiguity that is to be unpacked from the forest results in the selection of the actual most probable ambiguity in an overwhelming percentage of the time.

4.5.3. Unpacking the Highest Scoring Parse

Once the preliminary task of scoring all the nodes in the parse forest is complete, the actual parse disambiguation can be performed by a process of unpacking the best scoring analysis from the forest. While the node scoring was done in a "bottom-up" fashion, unpacking is done "top-down", starting from the node at the root of the forest. Unpacking is done by going down the parse tree, and selecting one of the alternative packed ambiguities at each packed parse node.

The unpacking process is guided by the scores of the parse nodes. At each packed node, the unpacking procedure attempts to select the alternative that received the highest probability. However, feature structure unification constraints are checked throughout the unpacking process, in order to guarantee that the unpacked analysis corresponds to a valid feature structure. This is done in the following way. After the unpacking of a packed node selects an ambiguity based on its score, the feature structure of the selected sub-analysis is temporarily assigned to the parse node.

This feature structure is then propagated up the parse tree, and all feature structure unifications at ancestor parse nodes are redone, in order to verify that the selection of the particular ambiguity at the current parse node is not ruled out by some higher level unification failure. If the unification checks done all the way back to the root node succeed, the selected sub-analysis is found to be consistent. The choice of which of the packed ambiguities was selected is then permanently recorded.

The unpacking process continues by moving on to unpack the children of the selected sub-analysis, one by one, left to right. After all of the children parse nodes are unpacked, a final unambiguous feature structure for the node is computed. The grammar rule corresponding to the disambiguated parse node is determined by the category of the node and its children. The feature structure unification associated with this grammar rule can then be re-executed, using the disambiguated feature structures associated with the children nodes. The resulting unambiguous feature structure for the parse node is then recorded.

If the unification consistency verification of a selected sub-analysis in a packed parse node fails, the sub-analysis is not part of a valid complete feature structure and must therefore be discarded. In this case, the unpacking procedure selects the next best scoring ambiguity in the packed node and repeats the unification consistency verification process. The highest scoring ambiguity that is found to also be unification consistent is eventually selected.

One should note that the fact that a particular choice at a packed parse node is found to be unification consistent implies that the subtree corresponding to this choice necessarily contains a sub-analysis that can lead to a complete parse tree. Therefore, the further unpacking of this chosen subtree is guaranteed to succeed, and no backtracking will ever have to be performed.

When unpacking terminates, an unambiguous feature structure is associated with the parse node at the root of the forest, and a corresponding parse tree can be constructed from the choices recorded at the packed nodes. Finally, the true probability of the unambiguous parse tree that was selected can be calculated, by a parse simulation procedure similar to the one used for training the probabilities. The parse tree is converted into an action sequence which is then used to simulate the sequence of transitions performed by the parser when the parse tree was created. The probability of the transition sequence is then computed and attached to the disambiguated parse as its true statistical score.

Limitations of the Unpacking Scheme

It is important to point out that our unpacking scheme has some limitations, that in certain cases hinder on it's ability to find and unpack the most probable analysis in the packed forest.

One problem, which we have already pointed out, is due to the fact that parse node scoring is a heuristic, which at times assigns overly optimistic scores to some of the parse nodes. Since the parse node scores determine which ambiguity is unpacked, there are instances where these score inaccuracies will result in the unpacking of a suboptimal analysis.

Another problem is due to the depth-first method by which unpacking is performed. Due to the effects of unification, the selection of a particular ambiguity at a packed node at one point in the unpacking process can rule out ambiguities at some other packed node that will be processed later on. This can result in the selection of a non-optimal analysis. The problem is demonstrated in Figure 4.4. The root of the forest is the node labeled `C`. The parse nodes labeled `A` and `B` are both packed nodes, where `A` packs ambiguities `A1` and `A2`, while `B` packs ambiguities `B1` and `B2`. Now assume the feature structures associated with the nodes are such that the unification at node

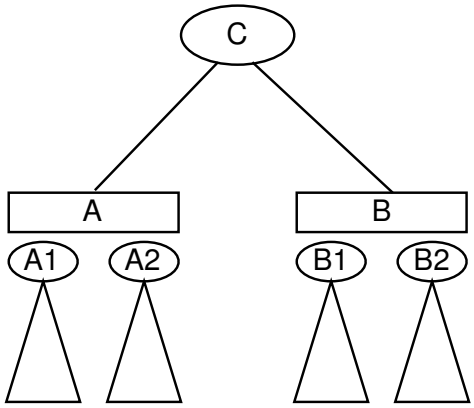


Figure 4.4 An Example of a Possible Sub-optimal Unpacking

C allows the combinations of A1 with B1 (but not with B2), and A2 with B2 (but not with B1). Furthermore, assume that A1 has a higher probability than A2, B2 has a higher probability than B1, and overall the combination of A2 with B2 results in an analysis of higher probability than that of A1 combined with B1. Our depth-first unpacking method will first visit the packed node A. Since A1 is more probable than A2, it will be processed first. Unification consistency checks will then be performed up the tree all the way to node C, to verify that A1 can lead to a valid parse. This check will succeed, since the unification at C allows A1 to be combined with B1. Since A1 is consistent, it is permanently selected by the unpacking procedure. When the procedure later on attends to the task of unpacking node B, B2 will be processed first, but it's consistency check will fail, since it can no longer be unified at node C with the feature structure forced by the selection of A1. Thus, B1 will have to be chosen as the selected ambiguity for node B. This results in the unpacking of the overall sub-optimal analysis that corresponds to A1 and B1.

One way to overcome the two above mentioned problems is to replace our disambiguation method with one that unpacks the parse forest in its entirety, computes the probability of each and every alternative analysis, and selects the truely highest scoring one. This is computationally expensive, and would defeat the purpose of performing ambiguity packing to begin with. We therefore chose to knowingly accept the limitations of the current unpacking scheme. Our experimental results have indicated that both of these problems are extremely rare in practice. In fact, we have yet to observe an instance in which the second problem actually occurs.

4.6. Evaluation

We have evaluated the statistical disambiguation module for two large practical grammars. The two grammars are drastically different in character. The first is a *syntactic* grammar developed for the ATIS domain. This grammar produces feature structures that represent the syntax of the input sentence. The second grammar is the Spanish analysis grammar developed for the JANUS/Enthusiast project. It is a predominantly semantic grammar (with some syntactic structural rules) for the appointment scheduling domain. The feature structures produced by the grammar are completely specified ILTs, representing the semantic relationships between the entities found

in the input sentence. The details of our performance evaluation in these two settings are described in the following subsections.

4.6.1. Statistical Disambiguation versus Minimal Attachment

In order to assess the effectiveness of our statistical disambiguation method, we need to compare its performance with some other well established disambiguation technique. Two common principle-based methods, Right Association and Minimal Attachment, were described earlier in the chapter.

As a criterion comparative with statistical disambiguation, we chose a method that is similar to Minimal Attachment. The classical notion of Minimal Attachment, as described earlier, makes the disambiguation decision "on-line", at the point where the ambiguous construct must be attached. Another possibility is to perform the disambiguation as a post-process, after the various complete parse trees have been constructed. In this case, Minimal Attachment corresponds to minimality in the total number of nodes in the parse tree. These two methods are usually (but not always) equivalent. Since we perform disambiguation as a post-process to parsing, we use this modified version of Minimal Attachment as a comparative disambiguation method.

It is easy to see that the total number of nodes in a parse tree is equal to the total number of actions (shifts and reduces) that the GLR parser takes in constructing the parse tree. Each terminal node in the tree corresponds to a shift action, while each non-terminal node corresponds to a reduce action. Thus, selecting a parse with a minimum number of parsing actions is equivalent to selecting a Minimal Attachment parse tree. The number of parsing actions of a particular parse tree is calculated by our statistical disambiguation module as a by-product of the statistical score. It was therefore easy to develop a Minimum Attachment disambiguation procedure, that selects among ambiguous parses using the value of the action counter instead of the statistical score.

A decision had to be made about what should be done in cases where multiple ambiguities happen to have the same total number of parse actions, and a preference cannot be established using Minimal Attachment. For the ATIS domain, we decided to break such ties using Right Association. This is implemented in the simple disambiguation procedure by comparing the parse action sequences of the alternative parse trees. Right Association corresponds to prefering a shift action over a reduce action. Thus, the preference is implemented by finding the first action in which the parse action sequences differ (i.e. the first shift/reduce "conflict"), and selecting the sequence in which a shift action was taken. For the Scheduling domain, Right Association did not seem to be an appropriate preference measure. Because the Scheduling grammars are *semantic* (and not syntactic), they were not designed in a way which would support correct disambiguation using Right Association. Therefore, for the Scheduling domain, we decided to use the fragmentation counter value as a secondary preference measure, in cases where Minimal Attachment cannot completely resolve the ambiguity. If the ambiguity is still not completely resolved, further ties are broken at random.

4.6.2. Evaluation on the ATIS Grammar

The ATIS grammar is a syntactic grammar for parsing sentences in the ATIS domain, where the language is limited to spontaneous spoken inquiries about commercial flights in the US. The core of this grammar is a general purpose syntactic grammar for English that was developed at the CMT. This core grammar consists of 364 rules, which provide a fairly wide coverage of common

syntactic structures of simple English sentences. Lexical analysis is provided by interfacing with a lexicon via a morphological package.

The core grammar was augmented by a relatively small set of domain dependent rules for the ATIS task. This set consists of 93 rules, most of which define the structure of ATIS specific entities such as flight numbers, airline and airport names, time expressions, etc. The complete grammar contains 457 rules, and compiles to a parse table of 688 states.

Since we did not have a collection of correct parses or feature structures for the ATIS grammar, we could not use the automatic training procedure for collecting the disambiguation statistics. We therefore used our semi-automatic interactive training procedure, where for each sentence, the trainer is required to select the correct analysis (feature structure) from the set of analyses produced by the parser. Because this task is time consuming, training was conducted on a set of only 500 sentences, from which a smoothed probability table was constructed.

We then tested the disambiguation module on an unseen test set of 119 sentences. For reference, this test set is included in Appendix B. 67 out of the 119 test sentences (56.3%) are ambiguous. In 50 out of these 67 ambiguous sentences (74.6%), one of the analyses produced by the parser is the correct analysis. The statistical disambiguation module was used to select an analysis for each of these ambiguous sentences. We then manually checked whether the analysis selected by disambiguation was the correct analysis. The disambiguation module selected the correct analysis in 37 out of the 50 sentences, resulting in a success rate of 74.0%. Using our combined Minimal Attachment/Right Association (MA-RA) disambiguation method on the same set of sentences, the parser was capable of selecting the correct parse in only 17 of the 50 sentences (34.0%). The statistical disambiguation method selected the correct ambiguity in 22 sentences where the MA-RA method failed. In 2 sentences, the MA-RA method was correct and the statistical method was wrong. The statistical method therefore out-performed the minimal attachment method by 20 sentences, a net improvement of 40.0%.

Analysis and Discussion

The ATIS grammar is highly ambiguous due to the fact that it is a general syntactic grammar. This is reflected by the fact that over 50% of the sentences in the test set were ambiguous, and required disambiguation. By far, the most common occurring type of ambiguity in the test set (and in the ATIS domain in general) is prepositional phrase (PP) attachment. It occurs in 46 out of the 67 ambiguous sentences of our test set (68.7%). For example, the sentence "*I need a flight from Pittsburgh to Boston*" has five different analyses, each corresponding to a different choice for PP attachment. Thus, the performance of the parser in the ATIS domain is highly dependent on the ability of our statistical disambiguation method to correctly resolve such PP attachment ambiguities.

It is important to note that the size of the training set that we used in this experiment was very small. Our parsing table contained 688 states, and a total of 14990 actions (with an average of about 22 actions per state). Training on the set of 500 sentences observed only 660 of the actions, in 275 of the states. Thus, our training observed a mere 4.4% of the possible actions, and no actions at all were observed for 60% of the states.

With such a small proportion of observed data by which we trained our model, one would expect the disambiguation to perform very poorly. However, our statistics work surprisingly well, particularly in resolving PP attachments. This is due to the fact that a significant number of

instances of frequently occurring ambiguities appeared even in our small training set. Because we train on *correct* parses, a particular preference can quickly be established, even from just a small set of training examples. In the case of PP attachment, a small set of training sentences similar to the sentence "*I need a flight from Pittsburgh to Boston*" is sufficient in order to establish the preference for attaching both PPs to the object noun ("flight" in this case). Consequently, out of 40 PP ambiguous test sentences for which there exists a correct parse, statistical disambiguation selects the correct analysis in 28 of the cases, a 70% success rate.

These results demonstrate that although using a much larger training set should result in significant improvements in the performance of statistical disambiguation, even extremely small amounts of *correct* training data can establish structural preferences for resolving the most frequently occurring types of ambiguity. Thus, significant benefits can be gained from using the statistical disambiguation module even when trained on relatively small amounts of training data. The size of the training set can be increased gradually over time, with an expected gradual increase in disambiguation performance.

It should also be noted that the combined MA-RA disambiguation method performed very poorly on the resolution of PP attachment ambiguities in the ATIS test set. The Minimal Attachment criterion did not distinguish among most of the these PP ambiguities. Furthermore, in most such cases, Right Association in fact prefered an incorrect analysis. For example, in the case of the sentence "*I need a flight from Pittsburgh to Boston*", Right Association chooses the analysis in which the PP "*to Boston*" is attached to "*Pittsburgh*" rather than to "*flight*". Further analysis showed that Minimal Attachment was unable to resolve the ambiguity (i.e several ambiguities, including the correct parse, had the same minimal number of parse actions) in 26 of the 50 ambiguous sentences in the test set. Right Association succeeded in selecting the correct ambiguity in only *one* of these 26 sentences. This resulted in the overall poor performance of the combined MA-RA disambiguation method.

4.6.3. Evaluation on the JANUS Spanish Analysis Grammar

The Spanish analysis grammar is a grammar that is being developed by Donna Gates at the CMT for the JANUS/Enthusiast project. For the purpose of this evaluation, we used the December-94 version of this grammar. The grammar contains 1107 rules and compiles into a parsing table with 2191 states. Lexical analysis is done by interfacing with a Spanish lexicon via a morphological package.

The probabilities for statistical disambiguation were trained on a corpus of "target" (correct) ILTs of 15 push-to-talk dialogs and 15 cross-talk dialogs, amounting to a total of 1687 sentences. This was done using the automatic training procedure described in section 4.4.3. We then evaluated the performance of the statistical disambiguation module on a unseen test set of 11 push-to-talk dialogs and 5 cross-talk dialogs, amounting to a total of 769 sentences. 524 out of the 769 sentences (68.1%) are parsable with one of the analyses returned by the parser being completely correct (thus matching the "target" ILT specified for the sentence). 213 of these 524 sentences (40.6%) are ambiguous. We applied the statistical disambiguation module to select an analysis for each of these ambiguous sentences, and compared the analysis picked by the disambiguation module with the "target" ILT for the sentence. The statistical disambiguation module successfully chose the correct parse in 140 out of the 213 sentences, thus resulting in a success rate of 65.7%. For comparison, we applied the Minimal Attachment disambiguation procedure to the same test set. With Minimal

Attachment, the correct parse was chosen in only 79 out of the 213 sentences, resulting in a success rate of 37.1%. Thus, the statistical disambiguation method outperforms the Minimal Attachment method in this case by about 29%.

Analysis and Discussion

The JANUS/Enthusiast Spanish analysis grammar is significantly less ambiguous than the ATIS grammar. This is mainly due to the fact that it is a semantic grammar, specifically designed to produce semantic structures in a very limited domain - appointment scheduling. Even so, a substantial portion of input sentences (about 40%) are ambiguous and require disambiguation.

It is interesting to note that statistical disambiguation appeared to be more successful in resolving certain types of structural and lexical ambiguities, and less successful in others. To a large extent, this may be a product of the amount of contextual information beyond the sentence level that is required for correctly resolving various types of ambiguity. Some examples of frequently occuring types of ambiguity in the Spanish scheduling domain can be seen in Table 4.2, along with a rating of how successful statistical disambiguation was in choosing the correct ambiguity for these types of ambiguities.

The size of the corpus that was available for training the probabilities of our statistical disambiguation module was about three times larger than the corpus used to train the probabilities for the ATIS grammar. Still, we manage to observe and model only a small fraction of the actual transitions allowed by the grammar. The parsing table for the Spanish grammar contained 2437 states, and a total of 43023 actions (with an average of about 18 actions per state). In the course of training the probabilities, only 2274 actions were observed, in 987 of the states. Thus, the training observed only 5.3% of all possible actions, and for 59% of the states, no actions were observed.

The statistical disambiguation results for the Spanish analysis grammar provide further evidence to our experience from the ATIS grammar, that even extremely small amounts of correct training data can establish structural preferences for resolving the most frequently occurring types of ambiguity. This once again points to the fact that significant benefits can be gained from using the statistical disambiguation module even when trained on relatively small amounts of training data.

Ambiguity	Examples	Number	Correctly Disambiguated by Statistical Disambiguation
Structural Ambiguity			
statement vs yes/no question	podemos reunirnos a las tres "we can meet at three" or "can we meet at three"	79 (27%)	65 (82%)
fixed-expression vs sentence	nos vemos "good-bye/see you", "let's see" or "we will meet"	11 (4%)	9 (82%)
How about that? vs What's up?	Qué tal "What's up" or "How about that"	15 (5%)	10 (67%)
busy or free	no tengo nada "I have nothing (planned)" or "I have nothing (free)"	4 (1%)	1 (25%)
temporal expressions	dos a cuatro por la tarde [two to four] in the afternoon two to [four in the afternoon]	24 (8%)	16 (67%)
date vs hour	de dos a cuatro "from two to four" or "from the second to the fourth"	34 (11%)	30 (88%)
miscellaneous		53 (18%)	35 (66%)
Lexical Ambiguity			
una	el lunes tengo *una* reunión "on Monday I have *a/one* meeting"	33 (11%)	31 (94%)
bueno	un tiempo bueno "a good time" or "a time ... well ..."	10 (3%)	4 (40%)
miscellaneous		35 (12%)	31 (89%)
Total		298 (282 sentences)	232 (78%)

Table 4.2 Statistical Disambiguation Performance on Several Common Types of Ambiguities in Spanish

Chapter 5

Parse Evaluation Heuristics

5.1. Introduction

At the end of the process of parsing a sentence, the GLR* parser returns with a set of possible parses, each corresponding to some grammatical subset of words of the input sentence. Due to the beam search heuristic and the ambiguity packing scheme, this set of parses is limited to maximal or close to maximal grammatical subsets. If the parser is to return a single parse result, it must select from among the set of found parses a parse that is deemed overall "best". Even in scenarios where the parser can pass on a set of possible analyses for further processing, it is very useful to be able to rank the set of parses found in terms of their plausibility. The primary criterion by which we wish to evaluate the parse results is maximality of word coverage. However, in many cases there are several distinct maximal parses, each consisting of a different subset of words of the original sentence. Thus, additional criteria must be employed in order to determine parse preferences. Furthermore, our experience has shown that in some cases, skipping an additional one or two input words may result in a parse that is syntactically and/or semantically more coherent.

The process of ranking the set of parses found by the parser and selecting the best parse can be viewed as an extension of the parse disambiguation process, which we described in the previous chapter. Whereas parse disambiguation attempted to select the best analysis from among a set of analyses of one particular parsed string of words, we are now faced with the extended problem of selecting among parses that correspond to different strings of words, each of which is a subset of the original input sentence. One of the metrics that can thus be used for evaluating the different parses is the statistical model developed for parse disambiguation.

The issue of how to best combine different parse evaluation measures has not been widely addressed in previous work. The few systems that have touched on this problem have focussed on combining evaluation measures in the context of parse disambiguation, and did not deal with the issue of selecting between parses of different input sequences. Linear combination appears to be the method of choice in several such systems. The proper set of weights for the various measures is typically determined manually by trial and error, or in a way that satisfies some intentional effects [34] [64]. Outstanding in this respect is the Spoken Language Translator (SLT), a joint research project of SRI, SICS and Telia Research [1], which uses a completely automatic two-stage process to determine the optimal set of evaluation measure weights. In the first stage, an initial set of weights is calculated using a least-square error criterion. An iterative "hill climbing" search procedure is then used to refine these initial weights by optimizing the performance on a training corpus.

This chapter describes an integrated framework that we have developed for the GLR* parser, that allows different heuristic parse evaluation measures to be combined into a single evaluation

function, by which sets of parse results can be scored, compared and ranked. The framework itself is designed to be general, and independent of any particular grammar or domain to which it is being applied. The framework allows different evaluation measures to be used for different tasks and domains. Additional evaluation measures can be developed and added to the system. The evaluation framework includes tools for optimizing the performance of the integrated evaluation function on a given training corpus.

We have also developed a set of concrete parse evaluation measures for the domains to which we have applied the GLR* parser. Although we believe that most of these measures should be useful in evaluating parses in other tasks and domains, we restrict ourselves to demonstrating their utility in our investigated settings.

The utility of a parser such as GLR* obviously depends on the semantic coherency of the parse results that it returns. Since the parser is designed to succeed in parsing almost any input, parsing success by itself can no longer provide a likely guarantee of such coherency. Although this task can in some cases be better handled by a domain dependent semantic analyzer that would follow the parser, we have developed a classification heuristic that is designed to try and identify situations in which even the "best" parse result is inadequate. This parse quality heuristic is described later in the chapter.

5.2. The Integrated Framework for Parse Evaluation

5.2.1. Evaluation Score Functions

The integrated parse evaluation framework is a method for combining different evaluation measures into a single evaluation function. Each of the available evaluation measures is called a *score function*. A score function can reflect any property of either a parse candidate or the skipped (or substituted) portions of the input that correspond to a parse candidate . The only restrictions we impose on score functions are the following:

1. The score function can be computed from the information available for each parse candidate at the end of parsing a given input.

2. The score function returns a positive real number.

3. The result returned by the score function is regarded as a *penalty*, so that a *lower* score corresponds to a preferable parse candidate.

5.2.2. Combining Score Functions

There are many different ways in which the set of evaluation score functions can be combined into a single evaluation score. Two simple possible combination schemes are to use either the product of the individual scores, or their sum.

Using the product of the scores is provably the correct combination scheme when all the evaluation measures are probability functions over the space of possible parses, and they are all independent. In this case, the product of the individual scores is a probability function that describes the joint probability of all the measures. However, we do not wish to restrict ourselves to evaluation

measures that are true probability functions, since some of our evaluation measures are difficult to model probabilistically. Furthermore, it is quite certain that the measures we use as score functions are not probabilistically independent.

We use *linear combination* as the scheme for combining the individual score functions. With linear combination, each of the individual scores are first scaled by a factor called a *weight*, and the results are then summed. It is easy to see that linear combination is a generalized version of the simple sum scheme mentioned above, since a simple sum corresponds to the case where all the weights are equal to one.

Linear combination provides a flexible and effective mechanism for combining the effects of different evaluation measures. The premise of this method is that there are instances in which one (or several) of the evaluation measures is better suited than the others in distinguishing which of the set of possible parses is preferable. Particularly in cases where several candidates may appear to be equally preferable according to one measure, a different measure can in effect "break the tie" in favor of one of the options. Linear combination also makes it possible to take into account the varying degree of significance of different evaluated properties. One or more of the properties can be given a dominant effect on the combined score function by being heavily weighted relative to the others.

5.2.3. Finding an Optimal Set of Weights

Selecting the weight assigned to each of the score functions allows us to intentionally enforce the dominance of certain evaluation measures over others. Thus, in contrast with the approach of the SLT project, we prefer to manually determine the initial set of weights assigned to the each of our score functions. However, the weights must be adjusted for every new task or domain to which the parser is applied. If done manually, this task must be performed by an expert, and even then there is no guarantee that the selected set of weights will achieve optimal or even near-optimal performance.

We have therefore developed a semi-automatic training procedure, which attempts to find the optimal combination weights for a given training corpus. This training procedure can be viewed as a search for a locally optimal function in the space of possible linear combinations. While not guaranteed to find a global optimal evaluation function, the training procedure uses "hill climbing" search to adjust the weights of the score functions in a way that monotonically increases the performance of the combined evaluation function. Our optimization procedure is similar in goal to the second stage procedure used by the SLT project [1], but the two procedures are quite different in their details.

Understanding the Effects of Weight Modification

The goal of the optimization procedure is to adjust the weights of the score functions in a way that maximizes the number of sentences in the training corpus for which the correct parse is assigned the best score. To understand how this can be done, let us assume the evaluation function for a parse p is a linear combination of k score functions, that has the form $S(p) = \sum_{i=1}^{k} c_i \cdot F_i(p)$, where F_i is the i-th score function, and c_i is its corresponding weight.

Let us now look at a case in which the correct parse did not receive the best score. To determine the reason why the correct parse did not receive the best score we need to compare the score values of

the correct parse with those of the parse that was ranked best. Let $S(p_{correct}) = \sum_{i=1}^{k} c_i \cdot F_i(p_{correct})$ be the score of the correct parse, and $S(p_{best}) = \sum_{i=1}^{k} c_i \cdot F_i(p_{best})$ be the score of the parse that was ranked best. We wish to compare the values $F_i(p_{correct})$ of the correct parse with the corresponding values $F_i(p_{best})$ of the parse that was ranked best.

An evaluation measure F_i for which $F_i(p_{correct}) > F_i(p_{best})$ is one that was a bad predictor in this case, since it tended to prefer the wrong parse. Lowering the weight of this score function might therefore improve performance. In the opposite case, a measure F_i for which $F_i(p_{correct}) < F_i(p_{best})$ is one that was a good predictor, since by itself it would have preferred the correct parse. Increasing the weight of this score function is thus likely to improve performance. In the case that $F_i(p_{correct}) = F_i(p_{best})$ the measure does not distinguish between the two parses, and nothing can be inferred about the weight assigned to the evaluation measure from this particular case.

Determining Which Weights to Modify

The information about how the evaluation scores of the correct parse compare with those of the best ranked parse must be collected on the entire set of sentences for which the correct parse was not ranked as best. We refer to this set of sentences as the *error set*. One possible way to do this is to collect the information on each of the scores F_i separately. A counter F_{i+} can keep track of the number of cases in which it appeared to be useful to increase the weight of the score function. A counter F_{i-} can keep track of the number of cases in which it appeared to be useful to decrease the weight of the score function. Once these counters are collected we can identify the measure for which $\Delta F_i = F_{i+} - F_{i-}$ has the greatest absolute value. Modifying the weight of this score function is likely to have the most significant effect on performance. If F_{i+} was greater than F_{i-}, the weight assigned to the measure should be increased. In the opposite case where F_{i-} was greater than F_{i+}, the weight of the measure should be decreased.

Collecting the counter information for each of the evaluation measures separately does not allow us, however, to take into account any interactions between the different measures. For example, it is possible that the most common case in the error set is one where the weight of one measure should be increased while at the same time the weight of another measure should be decreased. In order to capture such information, we maintain counters for *error vectors* instead of for the individual measures. An error vector is simply a vector of length n (where n is the number of evaluation measures), composed of "+" "−" and "0" signs. A "+" at position i in the vector indicates that the weight of F_i should be increased. Similarly, a "−" at position i indicates that the weight of F_i should be decreased, and a "0" indicates that the weight should not be changed. For example, for a set of four measures, the vector "+0−0" represents the combination of F_{1+} and F_{3-}, while F_2 and F_4 are to remain the same.

Note that for sentences where the best ranked parse is the correct parse, the values $F_i(p_{correct})$ and $F_i(p_{best})$ are identical, and the sentence can therefore be assigned to the vector "000$\cdots$". The error vectors can thus conveniently represent the results of analyzing the comparative information on evaluation scores for all sentences in the training set. For each sentence, we can compare the evaluation scores of the correct parse with those of the parse ranked best, and classify the sentence into one of the error vectors accordingly. The vector "000$\cdots$" should normally have the highest count, since it represents all the "good" cases, where the correct parse was ranked first. The other vectors are true error vectors.

We now pick the error vector that was most common. The vector indicates which weight modifications should prove to be most useful in improving performance. If position i in the vector is "+", the weight of F_i is increased. Similarly, if position i is "−", the weight of F_i is decreased. If position i is "0", the weight for F_i is not changed.

Weights are modified by discrete steps which are individually determined for each of the evaluation measures. It is important that the weight modification steps not be too large, since this may result in "jumping over" the local optima in the search space.

Modifying the weights according to the most common error vector will improve performance, if the new set of weights succeeds in changing the parse rankings of sentences that were classified to that vector in a way that the correct parse of some of these sentences is now ranked best. This is likely to be the case, since we increase the weights of evaluation measures that are good predictors, and decrease the weights of bad predictors. However, the weight modification is likely to also effect the parse ranking of sentences that were classified to other vectors, so the overall effect is not guaranteed to be better.

Some error vectors are unlikely to result in improved performance and are thus not considered in this process. We consider only error vectors that contain at least one "+" and one "−". This is because increasing the weights of evaluation measures according to an error vector that contains only "+" signs (and possibly some "0" signs) cannot change the ranking of parses of sentences that were assigned to this vector. Therefore, such weight modifications can only improve performance via some side effects on sentences classified to other vectors. A similar argument holds for error vectors that contain only "−" and "0" signs.

The Semi-automatic Optimization Procedure

The weight modification process that we have described above was developed into a semi-automatic iterative optimization procedure that can be viewed as performing a "hill climbing" search, starting with the manually pre-determined set of evaluation measure weights. In the first iteration, the procedure performs the following steps:

1. Parse all the sentences in the training corpus and rank the parse results using the evaluation function with the current weights.

2. Analyze the scores of the parses of each of the sentences, and classify the sentences to their corresponding error vectors.

3. Find the most common useful error vector, and determine discrete weight adjustments that correspond to the vector.

4. Update the weights of the evaluation function

In order to compare the evaluation scores of the correct parse of a sentence with those of the parse ranked as best, the correct parse must first be identified. This is done by comparing the ILTs of each of the resulting parses with a pre-defined correct "target" ILT. Once the correct parse is identified, the evaluation scores of the correct parse can be compared with the values of the parse ranked best, and the sentence can be classified into one of the error vectors.

In iterations subsequent to the first, the training set is reparsed using the new set of score function weights, the parses of the sentences are analyzed, and the sentences are again classified to

their proper error vectors. The results of this new classification must then be compared with those of the previous iteration, in order to determine if overall performance has improved. If performance did not improve, we go back to the old set of weights and may try modifying them according to the next most common error vector. If there are no more error vectors to try out, the process is halted, and the current set of weights are found to be locally optimal.

Our optimization procedure is only semi-automatic, due to the fact that it consults with the user before actually modifying the weights of the score functions, and proceeding to the next iteration. The user can decide whether to update the weights according to the most common error vector, some other error vector, or whether to halt at this point.

5.3. The Parse Evaluation Score Functions

The analysis of spontaneous speech has been the major task to which we have applied the GLR* parser. Within this task we have experimented with two different domains, the Scheduling domain and the ATIS domain. For these two domains we have developed a set of evaluations heuristics that is composed of the following four measures:

1. A penalty function for skipped words
2. A penalty function for substituted words
3. A penalty function for fragmentation of the parse analysis
4. A penalty function based on the statistical score attached to the parse

Although these measures were specifically developed for evaluating parses within the task of spontaneous speech analysis, we believe that with some adjustments, they should also prove to be useful in other tasks and domains. Furthermore, our parse evaluation framework allows us to easily augment these evaluation measures with new measures appropriate for the task and domain.

The details of these four evaluation heuristics are described in the following subsections.

5.3.1. The Penalty Function for Skipped Words

The most important of our parse evaluation heuristic measures is the penalty function for skipped words. Since we are primarily interested in parses of maximal (or close to maximal) coverage of the original input string, this penalty function is designed to prefer parses that correspond to fewer skipped words. In its most simple form, the function assigns a penalty of 1.0 for each word of the original string that had to be skipped. We will refer to this simple skipping penalty heuristic as *simple-skip*.

In order to try and cope with the phenomena of false starts and repeated words, which frequently occur in spontaneous speech, we modified the simple-skip heuristic to also account for the location in the sentence in which the skipped word occured. The intent is to assign a slightly smaller penalty for skipping words earlier in the input sentence. This will create a preference to chose a parse that skips a false start or the *first* of a pair of repeated words, since the penalty for such a skip will be slightly smaller. For example, consider the sentence "*I can um I can't meet you on Monday*". The two maximal parses found for this input correspond to the subsets "*I can meet you on Monday*"

and "*I can't meet you on Monday*". In both cases, two input words had to be skipped. However, we would like to prefer the second parse, since it corresponds to skipping the false start "*I can*".

The skip penalty dependency on location in the input is implemented in the following way. Instead of assigning a uniform penalty on 1.0 to each skipped word, we assign a penalty in the range of $[0.95, 1.05]$. Skipping the first word of the input string will incur a penalty of 0.95, while skipping the last word incurs a penalty of 1.05. The penalty for skipping a word anywhere between these two ends is determined by linear interpolation. We refer to this skipping penalty heuristic as *interpolated-simple-skip*.

Saliency Dependent Penalties for Skipped Words

The simple-skip and interpolated-simple skip heuristics operate under the basic assumption that all words are equally important and thus skipping over any word should incur the same penalty. This is obviously not the case, since in any domain, some words are more important to the correct interpretation of the sentence than others. Furthermore, skipping over certain function words, such as those indicating negation, can result in an analysis that is very different from the intended meaning, and should thus be penalized more heavily when skipped.

We therefore decided to develop a scheme in which the penalty for skipping a word would be some function of the saliency of the word in the domain. This scheme was implemented for the English scheduling domain (ESST) in the JANUS project. To determine the saliency of our vocabulary words, we compared their frequency of occurrence in transcribed domain text with the corresponding frequency in "general" text, estimated from a sample of text from the "Switchboard" corpus, a corpus of 2281 phone conversations between a pair of speakers. In precise terms, let $freq_{ESST}(w)$ be the frequency of word w in the ESST transcriptions, and let $freq_{SB}(w)$ be the frequency of w in the general text corpus. Then $sal(w)$, the saliency of word w, was defined as the ratio between these two frequencies, i.e.

$$sal(w) = \frac{freq_{ESST}(w)}{freq_{SB}(w)}$$

The saliency measure is then converted into a saliency dependent skip penalty by a linear transformation to the range $[0.0, 1.0]$. The word w_1, for which $sal(w_1)$ is lowest, is assigned a penalty value of 0.0, and the word w_2, for which $sal(w_2)$ is the highest, is assigned a penalty of 1.0. The penalty for words with a saliency that falls in between the two extremes is determined by linear interpolation.

As a final step, some manual post-processing was performed on the table of saliency dependent penalties, to address some anomalies, mostly due to the the effects of data sparseness. The skip penalties for certain classes of words, such as days-of-the-week, were made uniform by assigning each word in the class a penalty equal to the average penalty of words in the class. Also, the penalty assigned to negation words was increased, to reflect their relative importance in our application [1].

[1] words such as "*not*" are not significantly more frequent in our domain then in general text. Thus, the penalty value assigned to such words according to our saliency measure is rather small.We thus artificially inflate the penalty value for a small group of such words.

5.3.2. The Penalty Function for Substituted Words

In the context of a speech processing system, where the parser's input is the output of a speech recognizer, overcoming errors introduced by the recognizer is a major concern. Word substitutions are the most common type of such errors. We attempt to correct speech substitution errors by using the linguistic constraints imposed by the grammar in several different ways. These are described in detail in chapters 6 and 7. One of the methods involves using a confusion table. Words that are commonly misrecognized as other words are listed in the table along with one or more words that are likely to be their corresponding correct word. When working with a confusion table, whenever the GLR* parser encounters a possible substitution that appears in the table, it attempts to continue with both the original input word and the specified "correct" word(s).

Using a word substitution should incur some penalty, since we would like to choose an analysis that contains a substitution only in cases where the substitution resulted in a significantly better parse, due to the linguistic constraints of the grammar. Various methods can be used to assign the penalty for word substitutions.

We have so far experimented with using the confusion table method only for parsing speech recognized input in the ATIS domain. We currently use a simple penalty scheme for word substitutions, where each substitution is assigned a penalty of 1.0.

5.3.3. The Fragmentation Penalty Function

The nature of the spoken language in general, and spontaneous speech in particular, is such that utterances are frequently composed of grammatically incomplete sentences or pieces of sentences that we call *fragments*. Furthermore, due to limited grammatical coverage, the GLR* parser may not be able to parse a complete utterance even in cases where it is well structured. Nonetheless, most of the clauses in such utterances are likely to be parsable. The grammars we developed for parsing spontaneous speech using the GLR* parser were explicitly designed to tackle these problems. In order to successfully parse fragmented input, the grammars have very inclusive notions as to what may constitute a "grammatical" sentence. The grammars allow meaningful clauses and fragments to propagate up to the top (sentence) level of the grammar, so that fragments may be considered complete sentences. Additional grammar rules allow an utterance to be analyzed as a collection of several grammatical fragments.

In terms of their grammatical restrictness, our grammars for parsing spontaneous speech are thus rather "loose". The major negative consequence of this looseness is a significant increase in the degree of ambiguity of the grammar. In particular, utterances that can be analyzed as a single grammatical sentence, can often also be analyzed in various ways as collections of clauses and fragments. Our experiments have indicated that, in most such cases, a less fragmented analysis is more desirable. Thus, a mechanism for prefering less fragmented analyses was required.

One way in which this problem is indirectly addressed is via the statistical disambiguation module. The statistics collected for the grammar should implicitly reflect the preference for less fragmented analyses. However, because the statistical model accounts for only limited context, it is only partially effective in this respect. We therefore designed a separate explicit mechanism for expressing the fragmentation of an analysis, and for prefering analyses that are less fragmented.

The fragmentation of an analysis is reflected via a special "counter" slot in the constructed feature structure. The value of the counter slot is determined by explicit settings in the various

rules of grammar. This is done by unification equations in the grammar rule, that set the value of the counter slot in the feature structure that corresponds to the left-hand side non-terminal of the grammar rule. In this way, the counter slot can either be set to some desired value, or assigned a value that is a function of counter slot values of constituents on the right-hand side of the rule.

By assigning counter slot values to the feature structure produced by rules of the grammar, the grammar writer can explicitly express the expected measure of fragmentation that is associated with a particular grammar rule. For example, rules that combine fragments in less structured ways can be associated with higher counter values. As a result, analyses that are constructed using such rules will have higher counter values than those constructed with more structurally "grammatical" rules, reflecting the fact that they are more fragmented.

The preference for a less fragmented analysis is realized by using the value of the counter slot of the complete analyses as a parse evaluation measure. The value of the counter slot is viewed as a fragmentation penalty, since higher counter values reflect more fragmented analyses. The counter slot mechanism in effect allows the grammar writer to explicitly declare preferences between different grammar rules and sets of grammar rules. Although used to primarily reflect preferences with respect to fragmentation, the same mechanism can be used to express other preferences as well.

5.3.4. The Statistically Based Penalty Function

The statistical disambiguation module described in the previous chapter is based on a finite-state probabilistic model. As described earlier, the model's probability space consists of all possible state transition sequences of the parser, as defined by the LR parsing table. The probability of a parse analysis is determined from the state transition sequence to which the parse corresponds.

Most statistical disambiguation modules are designed to evaluate the probabilities of the set of analyses of a given input string. The probabilities of the various possible analyses of a particular input string are normalized, so that they sum to one. With such models, probabilities of parses of *different* input strings cannot be directly compared. Because we use a finite-state probabilistic model, the action probabilities do not depend on the given input sentence. The relative probabilities of the input words at various states are implicitly represented by the action probabilities. Because the finite-state probabilistic model assigns an actual probability to every transition sequence, transition sequences that correspond to parses of different inputs are directly comparable. This enables us to use the statistical score as a parse evaluation measure.

By using statistical parse scores as an evaluation measure across input strings, we hope to capture structural grammatical preferences, in the exact same way such preferences are used for parse disambiguation. For example, it may be the case that some subset of a particular input string is parsable, but only using a set of one or more uncommon grammar rules. On the other hand, a different subset of the same input string, corresponding to a different choice of skipped words, is also parsable, but with a set of more commonly used grammar rules. This difference should be reflected in the statistical scores of the parses of the two string subsets. Since each of the subsets may itself be ambiguous, we would like to compare the "best" statistical score of each of the alternative parsable substrings.

Because the process of unpacking the most probable ambiguity of a parse is computationally expensive, we wish to avoid having to actually do the unpacking for each of the parse candidates that correspond to different sets of skipped words. Instead, we use the estimated probability scores

that are computed for all the parse nodes by the bottom-up phase of the disambiguation process, as described in the previous chapter. Although these estimated scores are often somewhat over-optimistic, they are usually close enough to the correct probability of the best ambiguity to have the desired effect of prefering parses with a more common grammatical structure.

To enable parse probabilities to be used as an evaluation measure, they must first be converted into a penalty score, such that more "common" parse trees will receive a lower penalty score. The conversion to a penalty score should be such that the range of produced scores is comparable with the other evaluation scores, so that they can be combined with reasonable weight settings. We do this conversion using the following formula:

$$penalty = (0.1 * (-log_{10}(pscore)))$$

This conversion results in statistical penalty values that are most commonly in the range of 0.0 to 5.0, which puts them in the same order of magnitude as our other penalty scores.

5.4. The Parse Quality Heuristic

Since the GLR* parser is designed to find and parse grammatical subsets of a given input string, the parser fails altogether only on inputs where none of the subsets considered by the parser constitutes a grammatically complete sentence according to the grammar. Thus, it is expected that in most applications, a parsable subset will be found for almost any given input string. However, parsing success by itself does not provide any indication of how close in meaning the parse result is to the correct interpretation of the entire input utterance.

To address this problem, we have developed a classification heuristic that is designed to try and identify situations in which even the "best" parse result is inadequate. Our *parse quality heuristic* is designed to classify the parse chosen as best by the parser into one of two categories: "Good" or "Bad".

The classification is done by a judgement on the combined penalty score of the parse that was chosen as best. The parse is classified as "Good" if the value of the penalty score does not accede a threshold value. The threshold is a function of the length of the input utterance, since longer input utterances can be expected to accommodate more word skipping, while still producing a good analysis.

We currently use the classification procedure in Figure 5.1 for the scheduling domain in the JANUS speech translation project. The same classification procedure is used for both the English and Spanish analysis grammars. The calculated threshold is a sum of a threshold for the statistical penalty score and a threshold for the other three penalty scores. The threshold parameters were determined empirically. The statistical penalty threshold was calculated based on a linear regression estimate of the average statistical penalty as a function of the input length. A statistical penalty that is greater than the threshold is a good indication that the grammatical structure of the corresponding parse is uncommon, or was not observed in the training data. The threshold for the other three penalties was set so that their penalty sum would not accede 25% of the input length. Empirically this was determined to be a reasonable threshold. The two thresholds are combined together, to allow a good statistical score to compensate for bad other scores, and vice versa.

```
stat-threshold = 0.3 + (0.45 * length-of-input)
other-threshold = 0.25 * length-of-input

If (AND (penalty-score > 3.0)
        (penalty-score > stat-threshold + other-threshold))
then quality = ``Bad''
else quality = ``Good''
```

Figure 5.1 The Parse Quality Heuristic for the JANUS Scheduling Domain

5.5. Performance Evaluation

We evaluated the performance of the parse selection heuristics and the parse quality heuristic in two different settings. The first setting is an unseen test set of 513 transcribed Spanish dialogs in the scheduling domain, using the Spanish analysis grammar developed for the JANUS/Enthusiast project. The second setting is a unseen test set of 120 sentences in the ATIS domain, using the ATIS syntactic grammar.

The parse selection heuristics are evaluated by comparing the performance of the integrated set of parse scoring functions with a simple parse selection function that uses maximality of word coverage as the sole criterion for parse selection. In cases where there are several different maximal parses, the simple parse selection function chooses one at random.

The parse quality heuristic is evaluated by isolating the set of sentences that required skipping over portions of the input, judging by hand the quality of the parse selected as best, and comparing this manual judgement with the "Good/Bad" classification produced by the heuristic.

5.5.1. Evaluation on the JANUS Spanish Analysis Grammar

Training the Score Function Weights

The weights of the score functions were first trained on a training corpus of 635 English sentences in the scheduling domain, parsed with the English analysis grammar. These weights were then fine-tuned on a training corpus of 2390 Spanish sentences, parsed with the Spanish analysis grammar. The training was done using the semi-automatic weight adjusting procedure described earlier in the chapter. For the English analysis grammar we used the following three parse score functions:

1. The penalty function for skipped words, using saliency dependent penalties.

2. The fragmentation penalty function.

3. The statistical score penalty function.

As described earlier, the saliency dependent penalty for a skipped word is determined from the relative frequency of the word in in-domain versus out-of-domain texts, initially normalized to the range $[0.0, 1.0]$. Instead of adjusting the weight for the skipped words penalty function, we experimented with adjusting the range into which these penalties are normalized, which has a

similar but not identical effect. The initial weight of the statistical penalty score was 1.0, and the initial weight of the fragmentation penalty score was 0.5.

We then parsed the 635 English training sentences using these initial weights. This yielded a set of 21 sentences for which a correct parse exists but was not selected by the parser. These sentences were classified into their corresponding error vectors. 17 out of the 21 sentences were classified into the error vectors $+-+$ and $+--$, indicating that the weight of the penalty function for skipping words needed to be increased while the weight of the penalty function for fragmentation should be decreased. After a number of iterations of adjusting these two parameters, the fragmentation penalty weight was set to 0.25, and the skip penalty range was set to $[0.5, 5.0]$. With these settings, 14 of the 17 initial bad cases were fixed. Further attempts to tune the weights did not succeed in resolving the remaining 6 bad cases.

We re-parsed the entire training set again, to check the performance of the adjusted new weights. Only one additional new bad case was detected, while as expected, for 14 of the 21 previous bad cases, the parser now selected the correct parse. The final set of parameters chosen for the English analysis grammar are therefore:

1. The saliency dependent penalty for skipped words is normalized to a penalty in the range $[0.5, 5.0]$, with a weight of 1.0 .

2. The fragmentation penalty is factored by a weight of 0.25 .

3. The statistical score penalty is unfactored (i.e. a weight of 1.0).

The score function weights established for the English grammar were used as a starting point for the Spanish analysis grammar. For Spanish, we use the same set of parse scoring functions that were used for the English analysis grammar. However, we do not have saliency dependent penalties for skipped words in Spanish. The basic penalty for a skipped word is thus 1.0 . We suspected that the weights established for the English grammar would not be optimal for Spanish, and that some additional fine tuning would be required. In particular, it seemed likely that the weight of the penalty function for skipped words would have to be modified, since this penalty function for Spanish does not use saliency dependent scores.

We parsed a training corpus of 2390 Spanish sentences, with the Spanish analysis grammar using the score function weights from the English grammar. This produced a set of 21 sentences for which a correct parse exists but was not selected by the parser. It should be noted that the penalty score weights derived from the English grammar achieve surprisingly good results for the Spanish grammar, since they fail to pick the correct parse less than 1% of the time on the training set. Nevertheless, we continued to investigate if the weights could be further tuned.

The set of bad sentences was classified into error vectors. The most common error vector was $-0+$, with 7 cases. The vector $-++$ occurred 3 times and the vector $--+$ twice. Thus, 12 of the 21 cases indicate that the weight of the penalty function for skipped words should be decreased, while the weight of the statistical score penalty function should be increased. We experimented with several weight adjustments, but none of them managed to correct more than 4 out of the 21 error cases. This is not surprising, considering the already excellent performance of the previous weight settings. However, reparsing the entire training set revealed that any increase in the weight of the statistical score increases the overall number of bad cases on the entire training set. On the other hand decreasing the weight of the statistical score to 0.7 corrected 2 of the bad cases, and

	Parsable	Parsable with Matched ILT and Skipping	Correct Parse Selected	Unmatched Parse Marked "Good" with Skipping	Acceptable Translation Selected	Best Method Possible
Simple	500	33	24 (72.7%)	91	62 (68.1%)	74 (81.3%)
Full	500	33	26 (78.8%)	91	68 (74.7%)	74 (81.3%)

Table 5.1 Parse Selection Heuristics Performance Results

did not introduce any new errors. The final set of parameters for the Spanish analysis grammar is therefore:

1. The penalty for skipped words is in the range $[0.95, 1.05]$, with a weight of 1.0 .
2. The fragmentation penalty is factored by a weight of 0.25 .
3. The statistical score penalty is factored by a weight of 0.7 .

Evaluation Methodology

The performance of the parse selection heuristics was evaluated by two complementary methods. The first method compares the feature structure of the selected parse with a correct "target" ILT that is hand-constructed for the sentence being parsed. This method is similar to the one used in the process of training the score function weights, and can detect the sentences for which a completely correct parse exists but is not selected by the parser.

However, in the majority of sentences where parsing requires skipping over some portions of the input, none of the possible parses is completely correct, and thus comparing the resulting parses with a the correct ILT will not produce a complete match. For such sentences, a different method is required for evaluating if the selected parse is reasonably close to the intended meaning of the sentence. We accomplish this by generating an English translation from the selected parse ILT, and manually evaluating the quality of this translation. Cases where the translation is judged as bad are checked, in order to determine whether this is due to a generation error or to a bad parse ILT.

Performance Results - Parse Selection Heuristics

We evaluated the parse selection heuristics and the parse quality heuristic on an unseen test set of 513 Spanish sentences. The performance results are shown in Table 5.1. Our first evaluation method looked at complete matches with respect to a set of correct target ILTs. 500 out of the 513 sentences are parsable, and for 300 out of the 500 parsable sentences, one of the parses produced by the parser completely matches its corresponding target ILT. In 33 cases, the correct parse requires some word skipping. Using the simple parse evaluation heuristic, the *correct* parse was selected in 24 of the 33 cases (72.7%), while with the full integrated set of parse evaluation heuristics, 26 of the 33 cases (78.8%) were correct. The simple heuristic thus picks a wrong parse in 7 cases, while the full set of heuristics is wrong in only 5 cases. In 2 cases, the selected parse is correct (in terms of skipped words), but a wrong ambiguity was selected. Thus, the full set of parse evaluation heuristics performs only slightly better according to this evaluation method.

Parsable Marked "Bad"	With a Correct ILT	With an Acceptable Translation	With a Bad Translation	Parsable Marked "Good"	With a Correct ILT	With an Acceptable Translation	With a Bad Translation
57	0 (0.0%)	10 (17.5%)	47 (82.5%)	443	274 (61.9%)	389 (87.8%)	54 (12.2%)

Table 5.2 Parse Quality Heuristic Performance Results - Spanish Grammar

Our second evaluation method looked at the sentences for which no completely correct ILT is produced by the parser. We manually evaluated the quality of the translation of the these ILTs into English. 200 out of the 500 parsable test sentences do not produce a completely correct ILT. 148 out of the 200 are sentences that require some amount of skipping in order to be parsed. With the simple parse selection heuristic, the parser selects a parse that results in an acceptable translation in 72 out of the 148 sentences. With the full set of heuristics, an acceptable translation is produced in 78 out of the 148 cases.

It should be noted that in some of the 148 cases, all possible parses of the sentence are bad, and result in bad translations. Since it was infeasible to manually evaluate the translations of all possible parses of the sentences, it is difficult to estimate the number of additional sentences with parses that result in acceptable translations, beyond the 78 found by the full heuristics. However, since the parse quality heuristic is designed to filter out bad parses (see its separate evaluation below), we were able to look at the complete set of parses of sentences with skipping that were marked "Good", but for which the chosen parse is bad. The parse quality heuristic marks as "Good" 91 out of the 148 sentences. With the simple parse selection heuristic, acceptable translations are produced for 62 out of these 91 sentences, while with the full parse selection heuristics, 68 out of the 91 sentences result in acceptable translations. Thus, there are 23 sentences for which the parse chosen by the full heuristics is bad. After analyzing the translations produced by all possible parses of these 23 sentences, we found that in only 6 of these sentences there exists a parse that would produce an acceptable translation. Thus, a "best possible" parse selection method would be able to achieve acceptable translations for 74 of the 91 sentences (81.3%). Compared with this optimal method, our full heuristics selected a good parse in 68 cases (74.7%), while the simple heuristic selected a good parse in only 62 cases (68.1%).

Performance Results - Parse Quality Heuristic

The performance results of the parse quality heuristic are shown in Table 5.2. 57 out of the 500 parsable sentences are marked as "Bad" by the parse quality heuristic. We again evaluated the selected parse for these sentences, by comparing the parse with its corresponding target ILT, and by manually evaluating the English translation produced from the parse. None of the 57 sentences marked "Bad" have a completely correct parse that matches the corresponding target ILT. The selected parse for only 10 of the 57 sentences produces an acceptable translation, and for 47 out of the 57 sentences marked "Bad" (82.4%), the selected parse produces a bad translation. Of the 443 sentences marked "Good" by the parse quality heuristic, the selected parse for 274 of the sentences (61.9%) matches its target ILT, and for 115 of the sentences (26.0%), the selected parse is not a

```
If (AND (penalty-score > 3.0)
        (penalty-score > 0.5 * length-of-input)
then quality = ``Bad''
else quality = ``Good''
```

Figure 5.2 The Parse Quality Classification Procedure for the ATIS Grammar

complete match, but produces an acceptable English translation. Thus, the translation accuracy success rate for parses marked as "Good" by the parse quality heuristic is 87.9%.

These results demonstrate that the rather simple parse quality heuristic is fairly effective in detecting sentences that are parsable with the GLR* parser, but which do not produce an adequate parse result.

5.5.2. Evaluation on the ATIS Grammar

Setting the Parameters of the Heuristics

The grammar for the ATIS domain is a syntactic grammar that was developed from a general purpose syntactic grammar for English, using a development set of 300 sentences. The details of the grammar were described in the previous chapter in section 4.6.1. The parse evaluation functions used in conjunction with the ATIS grammar are the following:

1. The penalty for skipped words, where each skipped word receives a penalty in the range $[0.95, 1.05]$, depending on its position in the sentence.

2. The fragmentation penalty function.

3. The statistical score penalty function.

4. The substitution penalty function, where each substituted word receives a penalty of 1.0.

After some experimentation, the evaluation feature weights were set in the following way. The penalty score for skipped words was left unfactored, with a weight of 1.0. The penalty score for substituted words was assigned a weight of 0.9, so that substituting a word would be preferable to skipping the word. The fragmentation penalty function was given a weight of 1.1, to prefer skipping a word if it reduces the fragmentation count by at least one. The statistical score is unfactored, with a weight of 1.0.

Since we do not have a collection of "target" correct parses for the ATIS grammar, it is not possible to fine-tune these score function weights using the semi-automatic training procedure.

The parse quality classification procedure developed for the ATIS grammar appears in Figure 5.2. The parameters of this heuristic were set empirically based on experiments on the training set of 300 sentences.

	Parsable	Parsable with Skipping	Acceptable Parse Selected	Best Possible Method
Simple	119	53	16 (30.2%)	22 (41.5%)
Full	119	53	20 (37.7%)	22 (41.5%)

Table 5.3 Parse Selection Heuristics Performance Results - ATIS Grammar

Parsable Marked "Bad"	With an Acceptable Parse	With a Bad Parse	Parsable Marked "Good"	With an Acceptable Parse	With a Bad Parse
23	2 (8.7%)	21 (91.3%)	96	71 (74.0%)	25 (26.0%)

Table 5.4 Parse Quality Heuristic Performance Results - ATIS Grammar

Performance Results

We used a set of 120 new sentences as a test set. We compared the performance of the full heuristics described above with that of a simple heuristic that picks the parse with the least number of skipped and substituted words, breaking ties randomly. The parse selected best by the parser was evaluated by hand and categorized as either acceptable or bad. The results are shown in Table 5.3. 119 of the 120 sentences were parsable. The simple parse selection heuristic selects an acceptable parse for 69 out of the 119 parsable sentences (58.0%). The full parse selection heuristics select an acceptable parse in 4 additional cases, for a total of 73 out of 119 (61.3%). In both evaluations, the selected parse in 11 sentences is bad due to an incorrect ambiguity being picked. 53 of the 119 sentences required some degree of skipping in order to be parsed. With the simple heuristic, an acceptable parse was returned in 16 out of the 53 sentences that involved some degree of skipping. With our full heuristic scheme, an acceptable parse was returned in 20 sentences (4 additional sentences). Further analysis showed that only 2 sentences had parses that were better than those selected by our full parse evaluation heuristics.

We also evaluated the effectiveness of our parse quality heuristic in identifying bad parses. These results are shown in Table 5.4. 23 out of the 119 parsed sentences are marked as "Bad". The selected parse of only 2 of these 23 sentences (8.7%) is acceptable, while for 21 of the 23 sentences, the selected parse is indeed bad. 96 of the 119 parsed sentences are marked as "Good". The selected parse of 71 out of these 96 sentences (74.0%) is an acceptable parse. The selected parse for 25 of the 96 sentences (26.0%) is bad. For 11 of these sentences, the parse is bad solely due to a wrong ambiguity being picked. In only 14 cases (14.6%) did the parse quality heuristic fail to mark a truely bad parse as "Bad". Thus, the simple parse quality heuristic that we use for the ATIS grammar is quite effective in identifying parsable sentences that produce bad parses.

Chapter 6

Parsing Spontaneous Speech Using GLR*

6.1. Introduction

The analysis of spoken language is widely considered to be a more challenging task than the analysis of written text. All of the difficulties of written language (such as ambiguity) can generally be found in spoken language as well. However beyond these difficulties, there are additional problems that are specific to the nature of spoken language in general, and spontaneous speech in particular. The major additional issues that come to play in parsing spontaneous speech are speech disfluencies, the looser notion of grammaticality that is characteristic of spoken language, and the lack of clearly marked sentence boundaries. The contamination of the input with errors of the speech recognizer further exacerbates these problems.

A grammar based parser such as GLR*, that is designed to successfully process spontaneous speech, must address these issues via the constructed grammars, as well as the parsing architecture itself. While the particular grammars developed for spontaneous speech can concentrate on describing the structure of the meaningful clauses and sentences that are embedded in the spoken utterance, the parsing architecture must facilitate the extraction of these meaningful clauses from the utterance, while disregarding the surrounding disfluencies. The design of GLR*, as described in detail in the last three chapters, provides a parsing architecture that is well suited to handle the task of parsing spontaneous speech.

This chapter describes how the GLR* parser has been integrated into JANUS, a practical speech-to-speech translation project at Carnegie Mellon University. The system is designed to facilitate communication between a pair of different language speaking speakers, attempting to schedule a meeting. Thus, the current domain of the system is restricted to appointment scheduling. We describe how the GLR* parser was integrated as the parsing component of the system. We address the issue of how the parser is designed to handle the problem of parsing full speech utterances that lack explicit sentence boundaries. We then report on the performance of GLR* as an individual component within the system, and on the resulting overall end-to-end performance of the system. We draw conclusions about the robustness, accuracy and utility of GLR* for parsing spontaneous speech by a comparing the performance results of GLR* with similar evaluations using the non-robust original GLR parser and Phoenix, a robust parser designed for analyzing semantic concepts, which is available as an alternative parsing component in the JANUS system. We analyze the performance on both transcribed input and actual top-best hypotheses from the speech recognizer.

In the last section of the chapter, we evaluate the performance of GLR* on speech recognized input in the ATIS domain using a syntactic grammar. In the ATIS setting of the GLR* system,

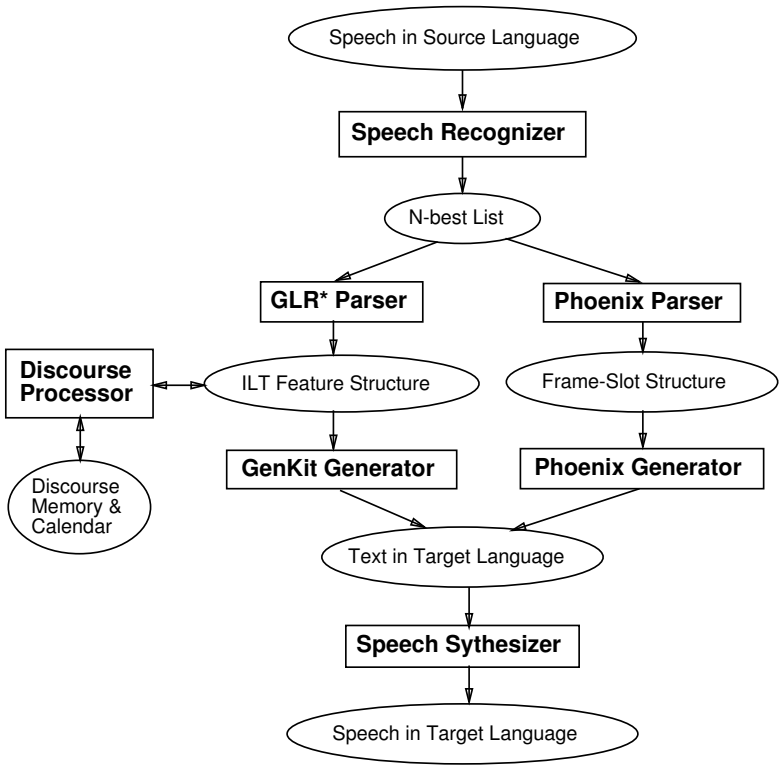


Figure 6.1 JANUS System Diagram

common occurring word substitutions of the speech recognizer are handled by using a simple word confusion matrix.

6.2. The JANUS Speech Translation System

JANUS [101] [92] [91] [106] [107] is a speech-to-speech translation system currently dealing with dialogs in the scheduling domain (two people scheduling a meeting with each other). The current source languages are English, German, and Spanish, and the target languages are English and German. Additional languages, such as Korean and Japanese are currently being added to the system. System development and testing is based on a collection of approximately 400 scheduling dialogs in each of the source languages.

The main modules of the JANUS system are speech recognition, parsing, discourse processing, and generation. Each module is designed to be language-independent in the sense that it consists of a general processor that applies independently specified knowledge about different languages. Therefore, each module actually consists of a processor and a set of language-specific knowledge sources. A diagram of the system is shown in Figure 6.1.

Processing starts with speech input in the source language. Recognition of the speech signal is done with acoustic modeling methods, constrained by a language model. The output of the speech recognizer is an n-best list of text strings representing the utterance hypotheses. In the current

configuration of the system, only the single top best hypothesis of the speech recognizer is passed on to the parser. We are also exploring a configuration of the system, in which the output of the speech recognizer is a word lattice, which is pre-processed and then passed on to a lattice parsing version of the GLR* parser. This version of the system is described in Chapter 7.

The top best speech hypothesis is then passed on to one of the two parsing components of the system. In principle, the processing tracks of both parsers can proceed in parallel, and be merged back together after the generation stage. However, we currently operate the system in configurations that incorporate only one of these two tracks. In the configuration in which the GLR* parser is used, the utterance is first pre-processed by a procedure that breaks long utterances into sub-utterances, that may each contain one or several sentences or clauses. The analysis grammars developed for the GLR* parser allow it to identify additional clause boundaries within each sub-utterance. The process by which this is done is described later in this chapter. The output of the GLR* parser is a set of interlingua texts, or ILTs, representing the semantic content and structure of the clauses of the utterance that were parsed by the parser. The GLR* parser can output either a single best disambiguated parse result, or a ranked list of possible analyses.

The discourse processor [81], based on Lambert's work [51] [52], disambiguates the speech act of each sentence, normalizes temporal expressions, and incorporates the sentence into a discourse plan tree. The discourse processor's focusing heuristics and plan operators eliminate some ambiguity by filtering out hypotheses that do not fit into the current discourse context. The discourse component also updates a calendar in the dynamic discourse memory to keep track of what the speakers have said about their schedules.

The updated set of ILTs is the sent to the GenKit generation module. GenKit [98] is a unification based generation system designed to complement the GLR parsing system. Similar to analysis grammars, generation grammars contain context-free rules augmented with LFG style feature unification operations. However, in the generation case, the feature operations constrain the generation of the string in the target language from a completely specified feature structure input. GenKit uses a top-down depth-first generation strategy. With well developed generation grammars, GenKit results in very accurate translation results for well specified ILTs.

In the alternative parsing path, the utterance is analyzed by the Phoenix parser. Phoenix is a parser specifically designed to extract semantic information which is then represented in frame-slot structures. The parser analyzes as much of the input utterance as possible by matching it with the patterns of the concept frames. The concept frame patterns are specified by recursive transition networks (RTNs). The parser can ignore words between slot-level concepts, but cannot ignore words interior to a concept pattern. It does not require any segmentation of the utterance into sentences or clauses. A more detailed description of the Phoenix parser is presented in the next section.

Since the frame-slot meaning representation produced by the Phoenix parser is different from the ILTs produced by the GLR* parser, target language output must be produced using a Phoenix generation module that is complementary to the parser. The Phoenix generator uses a simple left-to-right processing order to produce target language expressions from the analyzed concept tokens. The generation grammar consists of a set of target-language phrasings of each token, including lookup tables for such variables as numbers and days of the week. The result is a meaningful, but occasionally terse, translation.

In the final stage, the text in the target language, produced in either of the two possible language processing paths, is passed on to the speech synthesizer to produce a spoken utterance in the target

language. The JANUS project currently uses commercially available speech synthesizers for this task.

6.3. The Phoenix Parser

The Phoenix system [103] [63]) was designed for development of simple, robust natural language interfaces to applications. The Phoenix parser was originally developed for extracting semantic meaning representations from spoken utterances in the ATIS domain, where requests for flight information must be translated into appropriate database queries. The Phoenix system was ranked first in performance in a number of competitive DARPA ATIS evaluations.

The Phoenix parser uses frames to represent semantic relations. A frame represents some basic type of action for the application. Slots in a frame represent the information that is relevant to the action and are filled by matching patterns in an input string. The slots are filled independent of the order in which they appear in the frame. The patterns which fill slots are represented as Recursive Transition Networks (RTNs). Each slot has a grammar which specifies the patterns (strings of words) that can fill the slot. These grammars are called slot-nets. Since the slot-nets are compiled into RTNs, they can include calls to other networks.

In the course of parsing, the system uses slot-nets to match substrings in the input string. When a slot-net matches a substring, it is passed along for incorporation into frames. The system uses a beam search for frames. When a slot matches, it will extend all active frames that contain that slot. It will also activate any currently inactive frames that contain the slot.

The parser matches as much of the input utterance as possible to the patterns specified by the RTNs. Out-of-lexicon words are ignored. Words in the system lexicon, but not fitting the pattern being matched, will cause the slot pattern not to match. This does not cause the entire parse to fail, simply the slot being matched. The parser can ignore words between slot-level concepts, but cannot ignore words interior to a slot pattern. A version of the parser is under development which allows substitution, deletion and insertions in a pattern with a penalty.

The parser may combine slots together in any order, but in cases in which slot boundaries are not clear-cut it must decide how to segment the utterance. First, it looks for the interpretation with the most words matched. If there is no single best interpretation in this sense, it searches for the interpretation with the fewest number of slots. This is equivalent to finding the least fragmented analysis. If the interpretation is still ambiguous, it picks the one which has a fewer number of tokens at a higher level in the parse tree. Thus, an interpretation in which two tokens are nested is preferable to one in which they are sequential. The system has no real notion of sentence structure. The input is parsed as a simple sequence of frames. At the end of parsing the utterance, the single best parse found by the beam is returned.

The set of semantic tokens for the appointment scheduling task was developed from a set of 45 example English dialogues. The top-level frame tokens represent speech acts, such as suggestion or agreement. Lower-level slot tokens capture the specifics of the utterance, such as days of the week.

A typical `temporal` token can have as a sub-token a `date`, which can in turn consist of `month` and `day` sub-tokens. The `temporal` slot can fit into a frame representing a statement of unavailability, in which case a second slot suggesting an alternate time might follow.

```
Original utterance:
  THAT SATURDAY I'M NOT SURE ABOUT BUT YOU SAID YOU MAY BE BACK
  IF YOU THINK YOU'LL BE BACK THE THIS SUNDAY THE TWENTY EIGHTH
  I COULD SEE YOU AFTER ELEVEN AM ON THAT IF YOU'RE BACK

As decoded by the recognizer:
  *that saturday i'm not sure about but *you -said *you *maybe
  -back *into *think *to *be *back the sunday the twenty eighth
  i could see you after eleven am on *that *if *you -back

Parsed:
  [temporal] ( [point] ( [d_o_w] ( SATURDAY )))
  [give_info] ( [my_reluctance]
                ( I'M NOT SURE ABOUT ))
  [interject] ( [conj] ( BUT ))
  [give_info] ( [my_availability]
                ( [temporal] ( [point] ( THE
                [date] ( [d_o_w] ( SUNDAY ) THE
                          [day_num] TWENTY EIGHTH )))
                 I COULD SEE YOU ))
  [temporal] ( [range]
        ( [after] ( AFTER ) [time]
                  ( [hour] ( ELEVEN AM )) ON ))
```

Figure 6.2 Phoenix Analysis of a Typical Utterance

Figure 6.2 shows an example of a speaker utterance and the parse that was produced using the Phoenix system. The recognizer output, which is the text sent to the parser, is shown with unknown (−) and unexpected (*) words marked.

6.4. Parsing Full Utterances with GLR*

In order to be integrated into a complete speech-to-speech translation system, the parsing component must be capable of handling the type of data that is produced by the actual speech recognizer. The input to the speech system is in the form of a continuously uninterrupted spoken utterance from a user. In the context of a dialog between two users, such a utterance is called a *turn*. This utterance may consist of multiple clauses or sentences. The output of the speech recognizer is a textual hypothesis of the complete utterance, and contains no explicit representation of the boundaries between the clauses.

Our experience in analyzing spontaneous speech in the scheduling domain has shown that clauses form the fundamental "unit" of content. Although it is common to find some dependencies between clauses (mainly due to anaphora and ellipsis), the intended meaning can usually be well captured on the clausal level. Thus, the analysis grammars developed for the scheduling domain were constructed to primarily analyze clauses. Multi-clause utterances must therefore be broken

into clauses, in order to be properly parsed. The task of segmenting an utterance into clauses can be designated to a pre-processor to the parser, or it can be handled explicitly by the grammar in the course of parsing.

The approach we have taken in the JANUS system is a hybrid one, that combines both of these methods. The first and most straightforward step was to extend the analysis grammar with a high-level rule that allows the input utterance to be analyzed as a concatenation of clauses. However, the introduction of this simple rule results in some serious computational difficulties. This is due to the fact that for typical multi-clause utterances, there are many different possible ways of segmenting the utterance into clauses that are analyzable by the grammar. For long multi-clause utterances, the computational complexity of pursuing all such possibilities results in infeasible parse times and memory requirements. We significantly reduce the severity of this problem by using a pre-processor to break the utterance into smaller sub-utterances, that may still contain a number of clauses. In addition, we developed a powerful set of heuristics for further constraining the segmentations that are considered by the parser.

We call the segmentation of the utterance done by the pre-processor a *"hard"* break, since it forces the parser to consider each of the broken parts as separate units. This segmentation cannot be undone in the course of parsing. The segmentation performed by the parser, on the other hand, is called a *"soft"* break, since the parser has the liberty to decide on how to segment each parsed sub-utterance in a way that produces the best analysis.

The Utterance Segmentation Procedure

Although the output from the speech recognizer does not contain explicit clause boundaries, it does contain information that is highly indicative of such boundaries. Periods of silence and non-human noises that were detected by the recognizer appear as silence and pause "words" in the speech hypothesis. The recognizer distinguishes between various types of such non-content "words". These distinctions are not important to the parser, which considers them all as noise words.

The noise words in the speech hypothesis can be used in order to pre-break the speech recognized utterance. Because pre-breaking the utterance is a "hard" choice that cannot be corrected by the parser, we wish to avoid breaking the utterance at a point which in fact is not a clause boundary. On the other hand, it is acceptable to miss some of the actual clause boundaries, since the parser can later further segment the sub-utterances into clauses. The current criterion used by the pre-breaking procedure is the appearance of two or more consecutive noise words in the utterance. Our experience has shown that two consecutive noise words indicate a clause boundary in an overwhelming percent of the time. The appearance of single isolated noise words does not provide sufficient confidence for a clause boundary, since such noise words frequently occur within clauses as well.

We are currently working on enhancing this simple criterion for breaking the utterance. This work is being conducted in the context of developing an accurate procedure for breaking speech lattices (see Chapter 7), but will apply to the case of individual speech hypotheses as well. A minor modification is being made to the speech recognizer, so that information about the duration of silence and pause words will be explicitly represented in the speech hypothesis. Since long pauses are highly correlated with clause boundaries, the pause duration information can then be used to determine a threshold for breaking the utterance. Another possible source of information being investigated is the use of prosodic cues from the speech signal itself.

Constraining the Parser

The sub-utterances that are passed on to the parser may still contain several clauses. Using the top-level grammar rule that allows an utterance to consist of several clauses, the parser must determine the various ways in which the input can be segmented into clauses. However, to reduce the complexity of this task, we augmented the parser with several heuristics that constrain the set of clausal boundaries considered by the parser. The parser heuristics we have developed assume that the most desirable parse is one in which the utterance is segmented into as few clauses as possible. Since the grammar was developed to cover predominately individual clauses, this assumption generally holds well in practice (i.e. the grammar generally does not erroneously analyze multiple clauses as a single one). Our approach to the problem consists of two components. The first is a heuristic for pruning out all parses that are not minimal in the number of clauses. The second is a statistical method for blocking the parser from considering potential sentence boundaries at points in the utterance that are deemed to be statistically unlikely.

The pruning heuristic is implemented as a lisp procedure that is invoked along with the grammar rule that combines a new clause to a list of prior analyzed clauses. The feature structure associated with the list of prior analyzed clauses is pruned in a way that preserves only values that correspond to the minimum number of clauses. The f-structure of the new clause is then combined only with these selected values.

Since the clause combining grammar rule is invoked at each point where a part of the utterance may be analyzed as a separate clause, the pruning procedure incrementally restricts the parses being considered throughout the parsing of the input utterance. This results in a substantial decrease in the amount of memory required by the parser in the course of parsing the utterance.

The Statistical Clause Boundary Predictor

The second component of our solution is a procedure that restricts the points in the utterance that may be considered as clause boundaries by the parser, according to a confidence measure that is trained statistically. Specifically, the frequency of appearance of a clause boundary to the left, in between, and to the right of each pair of words (bigram) in the utterance is estimated from a training corpus. The procedure restricts the parser to consider clause boundaries only at points, where the combined probability of the surrounding bigrams exceeds a pre-determined threshold. This procedure was primarily developed and implemented by Noah Coccaro, a research programmer for the JANUS project.

The statistics are used to restrict the number of points in which the parser allows a clause boundary to occur. To achieve this, an additional rule that invokes the code was added to the grammar. Each clause that is parsed with the grammar must successfully "fire" this rule. The rule will only fire successfully if the point where the clause ends is a statistically reasonable point to segment the utterance. Should the rule fail, the parser will be prevented from pursuing a parse in which the following words in the utterance are interpreted as a new clause.

The grammar rule sends to the stochastic code the four words surrounding the point where the current clause would end (the two words prior to the point in question and the two words following it). Assume these are $[w_1 w_2 \bullet w_3 w_4]$, where the potential clause boundary is between w_2 and w_3. There are three statistics relevant to the decision of whether or not to allow a clause boundary at this point. These are:

1. $F([w_1 w_2 \bullet])$: the frequency of a clause boundary being to the right of the bigram $[w_1 w_2]$.

2. $F([w_2 \bullet w_3])$: the frequency of a clause boundary being in the middle of the bigram $[w_2 w_3]$.

3. $F([\bullet w_3 w_4])$: the frequency of a clause boundary being to the left of the bigram $[w_3 w_4]$.

These three frequencies are calculated from the number of times a clause boundary appeared in the training data in conjunction with the appropriate bigrams. In other words, if $C([w_i w_j \bullet])$ is the number of times that a clause boundary appears to the right of the bigram $[w_i w_j]$ and $C([w_i w_j])$ is the total number of times that the bigram $[w_i w_j]$ appears in the training set, then

$$F([w_i w_j \bullet]) = \frac{C([w_i w_j \bullet])}{C([w_i w_j])}$$

$F([w_i \bullet w_j])$ and $F([\bullet w_i w_j])$ are calculated in a similar fashion. However, for a given quadruple $[w_1 w_2 \bullet w_3 w_4]$, in order to determine whether the point in question is a reasonable place for breaking the utterance, we compute the following estimated frequency $\tilde{F}([w_1 w_2 \bullet w_3 w_4])$:

$$\tilde{F}([w_1 w_2 \bullet w_3 w_4]) = \frac{C([w_1 w_2 \bullet]) + C([w_2 \bullet w_3]) + C([\bullet w_3 w_4])}{C([w_1 w_2]) + C([w_2 w_3]) + C([w_3 w_4])}$$

This was shown to be more effective than the average or linear combination of the frequencies $F([w_1 w_2 \bullet])$, $F([w_2 \bullet w_3])$ and $F([\bullet w_3 w_4])$. The method we use is more effective because a bigram with a low frequency of appearance, for which we may not have sufficiently reliable information, is not counted as highly as the other factors.

In order for the grammar rule to be allowed to fire, the value of the computed fraction $\tilde{F}$ must be greater than a threshold set in advance. The value of the threshold is set manually so as to try and obtain the best performance on the training set. Each time the statistical check is evoked at parse time, there are four possible results:

1. The point in question is in fact a clause boundary, and the rule succeeds – a hit.

2. The point in question is in fact a clause boundary, and the rule fails – a miss.

3. The point in question is not a clause boundary, and the rule fails – a correct rejection.

4. The point in question is not a clause boundary, and the rule passes – a false alarm.

A correct rejection results in a saving of work by the parser, and an increase in efficiency. A false alarm does not damage the parse results, but is simply a lost opportunity to increase efficiency. A hit is an instance of the desired behavior, allowing the parser to break the utterance where it should be broken. A miss, on the other hand, inhibits the parser's ability to correctly parse the clauses in question, and will usually result in a bad analysis. We therefore tune the threshold such that there are as many correct rejections as possible, yet very few misses.

Test results of using the statistical clause boundary predictor on a test set of English scheduling data showed an overall decrease of about 30% in the time necessary to parse the test set. As an unexpected benefit, adding the statistical module improved parses of 15 utterances, while degrading only 7. This is contrary to intuition that performance is likely to be worse as a result of misses.

The performance improved because strange clause boundaries which the parser would previously have allowed are rejected, in favor of more reasonable locations.

To further improve the statistics, we used some word clustering. The clustering combines the statistics of words that have similar usage, thus increasing the amount of data we have on a bigram. Lists of clusters used in the speech language modelling were obtained and edited by hand. Words that were clustered but didn't seem to have similar usages were removed. Days, months, hours, and time of day are examples of clusters that were used.

6.5. JANUS Performance Evaluation

6.5.1. Evaluation Criteria and Methods

This section describes and analyzes the results of our performance evaluation of GLR* within the JANUS system. The evaluations reported in this section are designed to answer several different questions by which the performance of GLR* should be judged. The major questions being posed are the following:

1. **Robustness** - how robust is the GLR* parser to the types of disfluencies, noise and ill-formed input that occur in practice in spontaneous spoken scheduling dialogs that are processed by JANUS?

2. **Accuracy** - how accurate, and of what quality are the analyses that are produced using the GLR* parser, and what is the overall effect of the parser accuracy on the quality of the translations produced by the system?

3. **Utility** - How does GLR* compare with Phoenix, the alternative parsing architecture available in JANUS, and what are the strengths and weaknesses of each of the architectures?

We address the question of *robustness* by comparing the performance of GLR* with that of the original non-robust GLR parser from which it was developed. This test is easy to implement, since the word skipping capabilities of GLR* are controllable and can be completely disabled, resulting in a parser version that is equivalent to using the original GLR parser. We apply GLR and GLR* to a common unseen test set, using the same grammar.

The first measure we examine is how many of the test utterances were parsable by each of the two parsers. We then compare how many of the parsable utterances produced an acceptable translation. We also examine the ability of GLR* to mark bad parses using the parse quality heuristic, since in essence, this is equivalent to rejecting the utterance as unparsable. We thus look at how many of the utterances, that were marked "good" by the parser, had an acceptable translation. We also compare the precise correctness of the parse results of GLR and GLR*. This addresses the issues of both *robustness* and *accuracy*. For the Spanish test set, where a set of target ILTs was available, this is done using the ILT matcher. We analyze how many of the parser ILTs completely matched their corresponding target ILT. For the English test set, no target ILTs were available, and we thus analyze the precise translation quality of the parsed utterances.

To further address the question of *accuracy*, we examine how well the parse quality heuristic of GLR* was capable in identifying parses as "good" or "bad". We compare this with the parsability

test of of the original GLR parser, where any parsable utterance is considered "good", and any bad input fails to parse.

The question of *utility* is addressed by comparing the performance of GLR* and Phoenix on common test sets. Because the output produced by each of the parsers is different, we cannot compare these outputs directly, or by measuring them against a set of correct target ILTs. We therefore evaluate the parsers by comparing the quality of the English translations produced from their output. This measures the effect of parser performance on the end-to-end performance of the system. The procedures used for end-to-end evaluations are described in the next sub-section.

We attempt to substantiate our performance results with several evaluations using both Spanish and English as the source languages, and with transcribed text as well as top-best output from the speech recognizer as the form of input to the parser.

End-to-end Evaluation Methodology

Similar to the evaluations presented in the previous chapters, we use two different evaluation methods. The first method compares the ILT chosen by the parser with a pre-determined correct "target" ILT using an automatic matching program. This method can determine whether the ILT produced by the parser is completely correct. However, in many of the sentences where parsing requires skipping over some portions of the input, the parse result is reasonably close to the intended meaning of the sentence, yet not completely correct. We therefore use a complimentary end-to-end evaluation method, in which we manually evaluate the quality of the translation produced from the parser ILT. Although human involvement renders this method to be somewhat less objective, it is far better in evaluating parser performance in the context of the task at hand - spoken language translation.

End-to-end evaluation is conducted on an utterance level, assigning a grade to the translation of each input utterance. Each translation output is classified into one of three categories: "*Perfect*" "*OK*" or "*Bad*". When analyzing speech recognized data, we allow a fourth category, "*Recognition Error*". Bad translations that can be completely attributed to errors of the speech recognition are classified into this category. The criteria by which the grades are assigned are shown in Figure 6.3. Both "*Perfect*" and "*OK*" translations are considered acceptable, and are summed together when results are tabulated.

One of the drawbacks of relying on human judges to score translations is their subjectiveness. We have found that scores from different judges may vary by as much as 10 percentage points. The most reliable results are derived from employing a panel of at least three judges to score the translations. Their scores can then be averaged together. It is also preferable to use judges that are not directly involved in the development of the system knowledge sources. However, it is often difficult to meet these conditions. All of the end-to-end evaluation results reported in this section were graded by at least two judges. The Spanish evaluations were graded by Donna Gates and Marsal Gavalda, who develop the JANUS/Enthusiast Spanish analysis grammars for GLR* and Phoenix respectively. The English evaluations were graded by Kaori Shima and Laura Mayfield, who developed the JANUS English analysis grammars.

Perfect	Fluent translation with all information conveyed
OK	All important information translated correctly but some unimportant details missing or translation is awkward
Bad	Unacceptable translation

Figure 6.3 Evaluation Grade Categories

Evaluation on Grammatical vs. Ungrammatical Input

GLR* was designed to deal with two different types of phenomena: noise and disfluencies in the input, and limited grammar coverage. Both types frequently occur in spontaneous spoken language. Although the GLR* parser does not make a distinction between these two phenomena, it is useful and insightful to separately evaluate the capabilities of the parser in handling each of these problems. In order to do so, some additional evaluations were conducted on the JANUS English input test set. The test set was manually divided into two subsets - one consisting of input utterances that were judged to be completely grammatical, and the other consisting of utterances that contained noise, disfluencies or ungrammatical segments. In addition to evaluating parser performance on the entire set, we conducted similar evaluations on the two subsets. The results of these evaluations are presented later in this chapter.

Utterances taken from the "grammatical" subset and which cannot be fully parsed reflect grammar coverage limitations. Therefore, the performance evaluation on the "grammatical" subset allows us to assess how well GLR* copes with the problem of limited grammar coverage. Utterances in the "ungrammatical" subset contain noise and disfluencies, and will usually not fully parse. However, even if the utterance can be considered grammatical when stripped of its disfluencies, it may still not be fully parsable due to limited coverage of the grammar. Thus, to some extent, the evaluation on the "ungrammatical" subset still reflects the ability of the GLR* parser to deal with both types of phenomena.

6.5.2. Evaluation on Spanish Input

GLR* Compared with GLR

Table 6.1 and Table 6.2 compare the performance of GLR and GLR* on Spanish transcribed input. Speech recognized output for this test set was not available. The test set consisted of 11 push-to-talk and 5 cross-talk unseen transcribed Spanish dialogs that were pre-broken into sentences. The entire test set consists of 798 sentences (513 push-to-talk sentences from 89 utterances and 285 cross-talk sentences from 118 utterances).

The robustness results are presented in Table 6.1. While GLR is capable of parsing only 68.0% of the sentences, GLR* succeeds in parsing 97.9% of the sentences, an increase of almost 30%. Although this gain in parsable sentences is impressive, not all the additional parsable sentences result in correct parses or accurate translations. The fourth row in the table compares the number of parses for which an acceptable translation was generated from the ILT returned by the parser. For GLR, 515 sentences (64.5% of the total number of sentences) result in an acceptable translation. For GLR*, 628 sentences (78.7%) have an acceptable translation. The number of sentences with

	GLR		GLR*		Difference	
	number	percent	number	percent	number	percent
Total Test Set	798	100.0%	798	100.0%	-	-
Parsable	543	68.0%	781	97.9%	238	29.8%
Unparsable	255	32.0%	17	2.1%	238	29.8%
Parsable with Accept. Trans.	515	64.5%	628	78.7%	113	14.2%
Parsable Marked Good w/Accept. Trans.	515	64.5%	618	77.4%	103	12.9%
Matched Parses	442	55.4%	461	57.8%	19	2.4%
Unmatched Parses	101	12.7%	320	40.1%	219	27.4%

Table 6.1 Robustness Results: GLR vs. GLR* on Spanish Transcribed Input

	GLR		GLR*	
	number	percent	number	percent
Total Parsable Marked Good	543	100.0%	715	100.0%
Parsable Good w/Acceptable Translation	515	94.8%	618	86.4%
Parsable Marked Bad	0	-	66	100.0%
Parsable Bad w/Bad Translation	0	-	56	84.8%

Table 6.2 Accuracy Results: GLR vs. GLR* on Spanish Transcribed Input

an acceptable translation increased by 113 sentences (14.2%). Thus, about half of the additional sentences parsed by GLR* result in an acceptable translation, while the other half generate bad translations.

The parse quality heuristic of the GLR* parser attempts to detect such bad parses. For practical purposes, parses that are marked "Bad" by the parser should be considered similar to a parsing failure. It is thus meaningful to also look at the set of sentences that were marked "Good" by the parse quality heuristic of the parser. For GLR, all parsable sentences are considered marked "Good", since no word skipping is involved. The fifth row in Table 6.1 shows the number of sentences marked "Good" by the parser that have an acceptable translation. This result is 618 sentences for GLR* (77.4% of the total number of sentences), versus the 515 (64.5%) for GLR.

Thus, on parses marked "Good", GLR* still gets an additional 103 sentences with acceptable translations, which amounts to a 12.9% increase out of the total test set.

The last two rows of Table 6.1 examine the number of sentences for which the selected parse completely matched the specified target ILT for the sentence. For GLR, 442 sentences (55.4% of the total) completely match the target ILT. For GLR*, 461 sentences (57.8%) have a perfectly matching target ILT. The increase in perfect matches is therefore only 19 sentences, which amounts to 2.4% of the total test set. The number of parsable sentences that do not match their target ILT increases from 101 for GLR (12.7%) to 320 (40.1%). These results indicate that sentences that require word skipping in order to be parsed, most often result in a parse that is not completely correct, and thus does not completely match the target ILT. However, in roughly 50% of the time the missing information is not crucial to the translation of the sentence meaning, and the translation results are therefore acceptable.

The results in Table 6.2 analyze the accuracy of parse quality heuristic of GLR* on the test set, and compare them with GLR. The accuracy is measured by comparing the parser's judgement with respect to the quality of the parse, with the quality of the actual translation produced from the parse. For the GLR parser, parsability is the sole measure of accuracy, and any sentence that is parsable is considered to be good. GLR* uses the parse quality heuristic to determine whether the parse is acceptable. The first row in Table 6.2 is the total number of sentences for which the selected parse was considered good by each of the parsers. For GLR, the total is the 543 parsable sentences. For GLR*, the total is the 715 parsable sentences that were marked "Good". The second row in the table shows how many of the sentences considered good resulted in an acceptable translation. For GLR, 515 of the 543 parsable sentences (94.8%) have an acceptable translation. For GLR*, 618 of the 715 "Good" sentences (86.4%) have an acceptable translation. We also looked at the sentences that GLR* marked as "Bad". The parse of 66 sentences was marked "Bad" by GLR*. For 56 out of the 66 sentences (84.5%), the translation produced from the parse is in fact bad. GLR does not have a similar category of parses marked "Bad", since bad sentences are unparsable. However, of the 255 sentences that are unparsable by GLR (and thus judged as bad), 103 (40% of the 255) are parsed by GLR*, marked "Good" and have an acceptable translation. The results in Table 6.2 indicate that the GLR* parser's parse quality heuristic is about 85% accurate in classifying parses as either "Good" or "Bad", while GLR achieves an accuracy of about 95% on the sentences that it can parse.

GLR* Compared with Phoenix

Table 6.3 shows the results of a end-to-end performance evaluation that compared a version of JANUS that uses the GLR* parser with a version that uses the Phoenix parser. In this evaluation, the test set consisted of 2 cross-talk dialogs with a total of 54 utterances. Results on both transcribed input and speech recognized input are presented. For GLR*, both the transcribed and speech recognized utterances were pre-broken using the procedures described in section 6.4.

The end-to-end translation performance of both parsers on this test set was quite similar. On transcribed input, the percentage of utterances with acceptable translations using GLR* was slightly better than the corresponding figure when using the Phoenix parser (78.7% for GLR* versus 75.0% for Phoenix). Phoenix had a slight advantage on translations that were judged as "Perfect" (52.8% for Phoenix versus 51.9% for GLR*), but this was offset by a slightly larger advantage that GLR* had on translations judged as "OK" (26.9% for GLR* versus 22.2% for Phoenix).

	GLR*		Phoenix	
	Transcribed Input	Speech Recognized	Transcribed Input	Speech Recognized
Perfect Translation	51.9%	27.2%	52.8%	31.5%
OK Translation	26.9%	18.3%	22.2%	17.6%
Acceptable Translation	78.7%	45.5%	75.0%	49.1%
NIL Output	0.0%	0.0%	4.6%	1.9%
Bad Translation	21.3%	10.5%	20.3%	17.6%
Poor Recognition	-	44.0%	-	31.4%

Table 6.3 Performance Results of GLR* vs. Phoenix on Spanish Input (2 cross-talk dialogs, 54 utterances, 05-08-95)

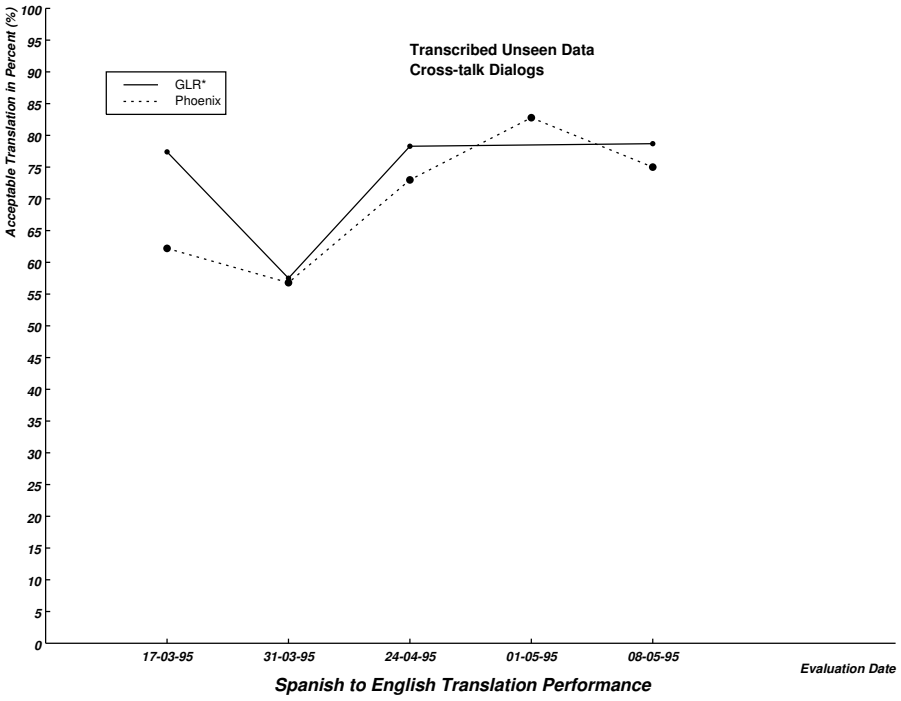


Figure 6.4 Sequence of GLR* and Phoenix Performance Evaluations - Unseen Transcribed Spanish Input

On speech input, where the system analyzes the top-best hypothesis produced by the speech recognizer, Phoenix performed slightly better than GLR*. The percentage of utterances with acceptable translations was 49.1% for Phoenix versus 45.5% for GLR*. Once again, Phoenix had slightly more "Perfect" translations (31.5% for Phoenix versus 27.2% for GLR*), and GLR* had slightly more "OK" translations (18.3% for GLR* versus 17.6% for Phoenix). For speech, bad translations that can be directly attributed to incorrect recognition are accounted for separately in the last row of the table. Poor recognition accounted for more of the bad translations when GLR* was used (44.0%) than when Phoenix was used (31.4%).

We have been conducting frequent end-to-end evaluations on unseen test sets in recent months, in order to assess the progress in grammar development. Figure 6.4 shows a comparative plot of the performance of GLR* and Phoenix over a sequence of several of these evaluations. The graph in the figure indicates the percent of utterances for which an acceptable translation was produced in each of the evaluations.

6.5.3. Evaluation on English Input

GLR* Compared with GLR

The robustness evaluation results, comparing GLR and GLR* on both transcribed and speech recognized English input are presented in Table 6.4 and Table 6.5. The test set consisted of 99 push-to-talk unseen dialogs. The transcribed text was pre-broken into clauses according to transcribed noise and pause markings. This produced a set of 360 sub-utterances or clauses, each of which was parsed separately by the parsers. The top-best speech recognized hypotheses were pre-broken using the two consecutive noise words criterion, as described in section 6.4. This produced a set of 192 sub-utterances or clauses, which were then parsed by the parsers.

On transcribed input (Table 6.4), GLR succeeds to parse 55.8% of the clauses, while GLR* succeeds in parsing 95.0% of the clauses. This amounts to a gain of about 39% in the number of parsable clauses. On the speech recognized input (Table 6.5), GLR parses only 21.4% of the input sub-utterances, while GLR* succeeds to parse 93.2%. In this case, the increase in parsable sub-utterances amounts to almost 72%.

The second part of the tables compares the number of parses for which an acceptable translation was generated. On transcribed input, GLR* produced acceptable translations for 85.6% of the clauses, compared with 54.2% for GLR. Thus, GLR* produces over 30% more acceptable translations. This is also the case when only the parses marked "Good" by the parse quality heuristic of GLR* are considered. The results on speech recognized input are rather similar. While GLR produced an acceptable translation only 17.2% of the time, GLR* did so in 47.9% of the time. Once again, this amounted to an increase of about 30.0% in the number of acceptable translations. Similar results were achieved when only the parses marked "Good" by GLR* are considered.

The third part of the tables examines the translation accuracy of GLR and GLR* on the test set. For transcribed input, perfect translations were achieved for 165 clauses (45.8%) for GLR, versus 212 clauses (58.9%) for GLR* (an increase of 13.1%). OK translations were produced for 30 clauses (8.3%) for GLR, versus 96 clauses (26.7%) for GLR* (an increase of 18.3%). GLR produced bad translations for only 6 clauses (1.7%), while GLR* did so for 34 clauses (9.4%), a difference of 7.7%. For the speech recognized input, the percent of perfect translations increased from 15.6% for GLR to 26.6% for GLR* (10.9% increase), and the percent of OK translations

	GLR		GLR*		Difference	
	number	percent	number	percent	number	percent
Total Clauses	360	100.0%	360	100.0%	-	-
Parsable	201	55.8%	342	95.0%	141	39.2%
Unparsable	159	44.2%	18	5.0%	141	39.2%
Parsable with Accept. Trans.	195	54.2%	308	85.6%	113	31.4%
Parsable Marked Good w/Accept. Trans.	195	54.2%	306	85.0%	111	30.8%
Perfect Trans.	165	45.8%	212	58.9%	47	13.1%
OK Translations	30	8.3%	96	26.7%	66	18.3%
Bad Translations	6	1.7%	34	9.4%	28	7.7%

Table 6.4 Robustness Results: GLR vs. GLR* on English Transcribed Input

	GLR		GLR*		Difference	
	number	percent	number	percent	number	percent
Total Clauses	192	100.0%	192	100.0%	-	-
Parsable	41	21.4%	179	93.2%	138	71.9%
Unparsable	151	78.6%	13	6.8%	138	71.9%
Parsable with Accept. Trans.	33	17.2%	92	47.9%	59	30.7%
Parsable Marked Good w/Accept. Trans.	33	17.2%	89	46.4%	56	29.2%
Perfect Trans.	30	15.6%	51	26.6%	21	10.9%
OK Translations	3	1.6%	41	21.4%	38	19.8%
Bad Translations	8	4.2%	87	45.3%	79	41.1%

Table 6.5 Robustness Results: GLR vs. GLR* on English Speech Input

increased from 1.6% for GLR to 21.4% for GLR* (19.8% increase). The percent of bad translations significantly increased from 4.2% for GLR, to 45.3% of the sub-utterances for GLR*. However, many of these bad translations are either due to errors of the speech recognizer, or are detectable by the parse quality heuristic (see next set of tables).

In summary, the robustness results on the English test set shows that using GLR* results in a 30% increase in the number of acceptable translations produced by the system, of which at least 10% are perfect translations, while the rest are not perfect but acceptable.

	GLR		GLR*	
	number	percent	number	percent
Total Parsable Marked Good	201	100.0%	330	100.0%
Parsable Good w/Acceptable Translation	195	97.0%	306	92.7%
Parsable Marked Bad	0	-	12	100.0%
Parsable Bad w/Bad Translation	0	-	10	83.3%

Table 6.6 Accuracy Results: GLR vs. GLR* on English Transcribed Input

	GLR		GLR*	
	number	percent	number	percent
Total Parsable Marked Good	41	100.0%	133	100.0%
Parsable Good w/Acceptable Translation	33	80.5%	89	66.9%
Parsable Good w/Acceptable Translation or Recognition Error	40	97.6%	128	96.2%
Parsable Marked Bad	0	-	46	100.0%
Parsable Bad w/Bad Translation	0	-	43	93.5%

Table 6.7 Accuracy Results: GLR vs. GLR* on English Speech Input

We next look at the effectiveness of the parse quality heuristic of the GLR* parser in identifying good and bad parses on this test set. These results are presented in Table 6.6 and Table 6.7. On the transcribed data (Table 6.6), 92.7% of the parses marked "Good" by the parse quality heuristic produce acceptable translations. This compares with 97.0% of the clauses parsable by GLR that result in acceptable translations. 10 out of 12 clauses marked by GLR* as "bad" (83.3%) result in a bad translation.

On the speech recognized data (Table 6.7), the effectiveness of the parse quality heuristic to identify bad parses is hindered by errors of the speech recognizer. Misrecognized times or dates, for example, can result in a poor translation, while not seriously affecting the parsability of the input. Thus, in total, parses marked as "Good" by the parse quality heuristic, produced acceptable translations for only 66.9% of the time (compared with 80.5% of the clauses parsable by GLR that produce acceptable translations). However, when errors that are completely due to poor recognition

are considered acceptable, 96.2% of the clauses marked "Good" by the parse quality heuristic are in fact acceptable (compared with 97.6% for GLR). Parses marked "Bad" by the parse quality heuristic produced a bad translation in 93.5% of the time.

Evaluation on Grammatical vs. Ungrammatical Input

The evaluation results presented so far demonstrated the effectiveness of GLR* in dealing with both noise and disfluencies in the input and limited grammar coverage. Although the GLR* parser does not make a distinction between these two phenomena, it is useful and insightful to separately evaluate the capabilities of the parser in handling each of these problems. In order to do so, additional evaluations were conducted on the JANUS English input test set.

The test set was manually divided into two subsets - one consisting of input utterances that were judged to be completely grammatical, and the other consisting of utterances that contained noise, disfluencies or ungrammatical segments. Of the 360 clauses in the transcribed test set, 249 clauses (69%) were judged as grammatical and 111 clauses (31%) were judged to be ungrammatical. The speech recognized input set of 191 sub-utterances was divided into a subset of 34 sub-utterances (18%) that were judged as grammatical and 157 sub-utterances (82%) judged as ungrammatical.

Table 6.8 and Table 6.9 show the results of our evaluation on the grammatical and ungrammatical subsets for transcribed input. On the grammatical subset (Table 6.8), GLR parses 69.5% of the clauses, while GLR* parses 96.8% of the clauses. Thus, about 30% of the grammatical clauses cannot be fully parsed, due to grammar coverage limitations. GLR* succeeds in parsing all but 3% of these clauses (a gain of about 27%). On the ungrammatical subset (Table 6.9), GLR parses only 25.2% of the clauses, while GLR* succeeds to parse 91.0%. In this case, the increase in parsable clauses amounts to almost 66%. One would expect that none of the ungrammatical clauses would be parsable by GLR. Yet 25% of the clauses are parsable in full. This is due to the fact that our semantic grammar for the scheduling domain is flexible and tolerates some amount of ungrammaticality. However, indeed most of the clauses (about 75%) cannot be parsed in full. By skipping over some portion of the input, GLR* succeeds in finding a parse for all but 9% of these clauses.

The second part of the tables compares the number of parses for which an acceptable translation was generated. On the grammatical subset, GLR* produced acceptable translations for 90.0% of the clauses, compared with 67.5% for GLR. Thus, by overcoming limitations in the grammar coverage, GLR* produces over 22% more acceptable translations. This is also the case when only the parses marked "Good" by the parse quality heuristic of GLR* are considered. The results on the ungrammatical subset are rather similar. While GLR produced an acceptable translation only 24.3% of the time, GLR* did so in 51.4% of the time. Thus, by overcoming noise and speech disfluencies (and some grammar coverage limitations), GLR* provides about 25% more acceptable translations than GLR. The results are similar when only the parses marked "Good" by GLR* are considered.

The third part of the tables examines the translation accuracy of GLR and GLR* on the test set. For the grammatical subset, perfect translations were achieved for 148 clauses (59.4%) for GLR, versus 170 clauses (68.3%) for GLR* (an increase of 8.9%). OK translations were produced for 20 clauses (8.0%) for GLR, versus 54 clauses (21.7%) for GLR* (an increase of 13.7%). GLR produced bad translations for only 5 clauses (2.0%), while GLR* did so for 17 clauses (6.8%), a difference of 4.8%. For the ungrammatical subset, the percent of perfect translations increased

	GLR		GLR*		Difference	
	number	percent	number	percent	number	percent
Total Clauses	249	100.0%	249	100.0%	-	-
Parsable	173	69.5%	241	96.8%	68	27.3%
Unparsable	76	30.5%	8	3.2%	68	27.3%
Parsable with Accept. Trans.	168	67.5%	224	90.0%	56	22.5%
Parsable Marked Good w/Accept. Trans.	168	67.5%	223	89.6%	55	22.1%
Perfect Trans.	148	59.4%	170	68.3%	22	8.9%
OK Translations	20	8.0%	54	21.7%	34	13.7%
Bad Translations	5	2.0%	17	6.8%	12	4.8%

Table 6.8 GLR vs. GLR* on Grammatical English Transcribed Input

	GLR		GLR*		Difference	
	number	percent	number	percent	number	percent
Total Clauses	111	100.0%	111	100.0%	-	-
Parsable	28	25.2%	101	91.0%	73	65.8%
Unparsable	83	74.8%	10	9.0%	73	65.8%
Parsable with Accept. Trans.	27	24.3%	84	75.7%	57	51.4%
Parsable Marked Good w/Accept. Trans.	27	24.3%	83	74.8%	56	50.5%
Perfect Trans.	17	15.3%	42	37.8%	25	22.5%
OK Translations	10	9.0%	42	37.8%	32	28.8%
Bad Translations	1	0.9%	17	15.3%	16	14.4%

Table 6.9 GLR vs. GLR* on Ungrammatical English Transcribed Input

from 15.3% for GLR to 37.8% for GLR* (22.5% increase), and the percent of OK translations increased from 9.0% for GLR to 37.8% for GLR* (28.8% increase). The percent of bad translations increased from 0.9% for GLR, to 15.3% of the clauses for GLR*. However, as seen earlier, many of the bad translations are detectable by the parse quality heuristic.

Table 6.10 and Table 6.11 show the results of our evaluation on the grammatical and ungrammatical subsets for speech recognized input. On speech recognized input, a much larger portion of the test set (82%) belongs to the ungrammatical subset. This is due to errors of the speech recognizer that corrupt many of the utterances. On the grammatical subset, using GLR* results in about

	GLR		GLR*		Difference	
	number	percent	number	percent	number	percent
Total Clauses	34	100.0%	34	100.0%	-	-
Parsable	27	79.4%	34	100.0%	7	20.6%
Unparsable	7	20.6%	0	0.0%	7	20.6%
Parsable with Accept. Trans.	26	76.5%	33	97.1%	7	20.6%
Perfect Trans.	26	76.5%	30	88.2%	4	11.7%
OK Translations	0	0.0%	3	8.8%	3	8.8%
Bad Translations	1	2.9%	1	2.9%	0	0.0%

Table 6.10 GLR vs. GLR* on Grammatical English Speech Recognized Input

	GLR		GLR*		Difference	
	number	percent	number	percent	number	percent
Total Clauses	157	100.0%	157	100.0%	-	-
Parsable	12	7.6%	144	91.7%	132	84.1%
Unparsable	145	92.4%	13	8.3%	132	84.1%
Parsable with Accept. Trans.	6	3.8%	59	37.6%	53	33.8%
Perfect Trans.	4	2.5%	21	13.4%	17	10.9%
OK Translations	2	1.3%	38	24.2%	36	22.9%
Bad Translations	6	3.8%	85	54.1%	79	50.3%

Table 6.11 GLR vs. GLR* on Ungrammatical English Speech Recognized Input

a 21% gain in both parsable sub-utterances and acceptable translations. On the ungrammatical subset, the gain in parsable sub-utterances is over 84%, while acceptable translations increase by almost 34%.

GLR* Compared with Phoenix

Table 6.12 shows the results of the end-to-end performance evaluation that compared GLR* with the Phoenix parser on English transcribed and speech data. The same test set of 99 utterances that was used in the GLR versus GLR* evaluation was used. Both GLR* and Phoenix used the same pre-breaking methods. End-to-end translation results were evaluated on a complete utterance level.

The end-to-end translation performance of GLR* and Phoenix on the English test set was quite similar. On both transcribed and speech input, the percentage of utterances with acceptable translations using Phoenix was slightly better than the corresponding figure when using GLR*. On transcribed input, GLR* produces acceptable translations for 83.9% of the utterances, versus 85.5% for Phoenix. On speech input, GLR* has acceptable translations for 44.5% of the utterances, while

	GLR*		Phoenix	
	Transcribed Input	Speech Recognized	Transcribed Input	Speech Recognized
Parsable	100.0%	100.0%	99.0%	100.0%
Unparsable	0.0%	0.0%	1.0%	0.0%
Perfect Translation	45.5%	18.2%	48.0%	19.8%
OK Translation	38.4%	26.3%	37.5%	26.0%
Acceptable Translation	83.9%	44.5%	85.5%	45.8%
Bad Translation	16.1%	12.1%	13.5%	11.5%
Poor Recognition	-	43.4%	-	42.7%

Table 6.12 Performance Results of GLR* vs. Phoenix on English Input (14 dialogs, 99 utterances, 07-14-95)

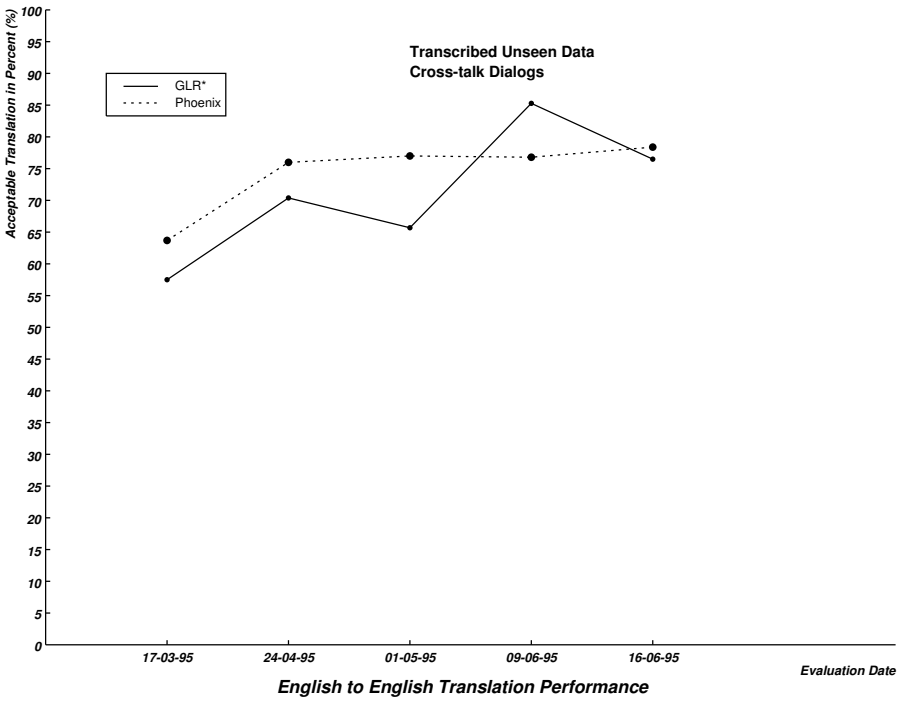


Figure 6.5 Sequence of GLR* and Phoenix Performance Evaluations - Unseen Transcribed English Input

Phoenix has acceptable translations for 45.8%. For both transcribed and speech input, Phoenix had slightly more translations that were judged as "Perfect", while GLR* had slightly more translations judged as "OK".

Figure 6.5 shows a comparative plot of the performance of GLR* and Phoenix over a sequence of several recent evaluations on transcribed input. The graph in the figure indicates the percent of utterances for which an acceptable translation was produced in each of the evaluations.

6.5.4. Discussion of Results

Robustness

The analysis of the robustness evaluations, that compared GLR* with the original GLR parser, demonstrate that GLR* is far better suited to the parsing of both transcribed and recognized spontaneous speech of the kind processed by the JANUS system. The most meaningful measure of performance is the percentage of acceptable end-to-end translations that are produced by the system with each of the parsers. On transcribed input, GLR* produced 13% more acceptable translations than GLR on the Spanish test set, and 31% more acceptable translations on the English test set. A similar increase of about 30% was measured on English speech recognized input. Most of these additional acceptable translations are not perfect. On the Spanish test set, an increase of only 2.5% in the number of target ILT matches was measured. On the English transcribed test set, the number of perfect translations increased by about 13%, and on the English speech set it increased by about 11%. Thus, although the vast majority of the additional acceptable translations found by GLR* are not perfect or complete, they still reflect the fundamental meaning being conveyed by the speaker. Cases where the portions of the input that were skipped by GLR* are important to the meaning of the utterance can be detected quite effectively by the parse quality heuristic that is used in conjunction with GLR*.

The additional evaluations on grammatical versus ungrammatical subsets of the English test set provided further insight on the capabilities of the GLR* parser. The performance results on the grammatical subset demonstrate that GLR* is very effective in overcoming the problem of limited grammar coverage. The percentage of acceptable translations for transcribed grammatical clauses increased from 68% using GLR, to almost 90% using GLR*. While the grammar fails to fully cover about one third of the clauses in the test set, GLR* obtains a good "approximate" parse for many of these clauses, by skipping over minor portions of the input. The segments that are skipped often have only minor semantic significance and are not crucial to the translation task at hand. Only in 10% of the clauses did the skipping performed by GLR* result in an unacceptable translation. This appears to indicate that using GLR* in conjunction with a grammar of reasonably adequate coverage can result in very high levels of performance, where minor unimportant portions of the input that are not covered by the grammar are detected by the parser and ignored. Thus, GLR* should prove to be very useful in overcoming grammar coverage problems in other tasks, where input is for the most part grammatical, but complete parsability is not crucial.

The performance results on the subset of ungrammatical clauses highlight the robustness that GLR* provides against the noise and disfluencies that are typical of spontaneously spoken language. Almost one third of the transcribed clauses were judged as ungrammatical, thus containing some form of noise or disfluency. While GLR produces an acceptable translation for only 24% of these clauses, GLR* impressively does so for almost 76% of the clauses. These results indicate that most

of the ungrammatical utterances in our domain contain largely grammatical structures, sparsely surrounded by noise and disfluencies. GLR* is very successful in ignoring these phenomena and extracting the underlying grammatical structure that reflects the meaning of the utterance.

It is also interesting to compare the performance results of GLR* on transcribed input with the corresponding results on speech recognized input. At first glance, the speech performance results appear to be rather disappointing. Speech recognition errors often prohibit the parser from finding a correct analysis for the input utterance, and this results in far lower performance on speech recognized input. Whereas for transcribed input, GLR* produced an acceptable end-to-end translation in about 85% of the time on the unseen English test set, the comparable performance with speech recognized input on the same test set was only 46%. Furthermore, a speech recognition error was identified as the primary cause of a parsing failure in about 87% of all bad parses. However, a separate analysis of the performance of the speech recognizer on this test set revealed that the word accuracy rate was about 65%, and of the 99 utterances in the test set, only 4 (rather short) utterances were completely free of recognition errors. Consequently, over 80% of the English test set speech-recognized sub-utterances were judged as ungrammatical, mostly due to speech recognition errors. While the GLR parser is virtually incapable of handling any of these sub-utterances (producing an acceptable translation less than 4% of the time), GLR* produced an acceptable translation for almost 38% of the ungrammatical speech recognized sub-utterances. This indicates that while overcoming speech recognition errors is a major problem that must be addressed (see Chapter 7), the GLR* parser's robustness to noise and disfluencies provides a very significant performance boost.

Accuracy

Because of the word skipping nature of GLR*, the quality of the parses that are found by the parser is bound to be somewhat lower than the quality of the parses found by GLR, where parsable input is completely grammatical. The parse quality heuristic that is used with GLR* turns out to be fairly effective in detecting parses where important portions of the input were skipped by the parser. Parses marked as "Bad" by the heuristic are indeed not acceptable in 85% of the time on Spanish transcribed input, 83.3% on English transcribed, and 93.5% on English speech input. Whereas the portion of acceptable translations out of the transcribed utterances parsable by GLR was 95% on Spanish and 97% on English, the equivalent portions of acceptable parses out of the parses marked "Good" by GLR* was 86% on Spanish and 93% on English. On English speech input, 96% of the parses marked "Good" are either acceptable, or are bad solely due to errors of the speech recognizer.

The differences between GLR and GLR* in robustness and accuracy can be viewed as a classic example of the tradeoff between precision and recall. In order to achieve the higher level of accuracy (precision), GLR must sacrifice a large degree of robustness (recall). By using the parse quality heuristic, we try to strike a reasonable balance in this tradeoff.

Utility

The evaluations comparing GLR* with Phoenix on both the Spanish and English test sets indicate that the portion of acceptable translations produced with each of the parsers is very similar. On the Spanish transcribed test set, GLR* is slightly better (78.7% versus 75% for Phoenix), while on

English transcribed data, Phoenix is slightly better (85.5% versus 83.9% for GLR*). On speech recognized data, Phoenix performed slightly better than GLR* in both Spanish and English. It should be noted that these slight performance differences should not be regarded as statistically significant. The translation quality evaluations are conducted manually by the grammar developers of each of the parsers, and are thus subjective. Although the grading is cross validated (the two grammar developers for each language grade the output from both GLR* and Phoenix, and the results are then averaged), subjective differences in judgement between the graders have repeatedly amounted to 5% or more.

However, the two parsing architectures do have some clear strengths and weaknesses. Although not quite evident from the evaluations reported here, tests conducted during earlier stages of the development of the grammars appeared to indicate that even though the total portion of acceptable translations produced by both parsers is similar, GLR* tended to produce more perfect translations, while phoenix produced more translations that were merely OK. This appeared to indicate that GLR* was somewhat more accurate and Phoenix was somewhat more robust.

GLR* tends to break down when parsing long utterances that are highly disfluent, or that significantly deviate from the grammar. In many such cases, GLR* succeeds to parse only a small fragment of the entire utterance, and important input segments end up being skipped. Phoenix is significantly better in analyzing such utterances. Because Phoenix is a chart parser that is capable of skipping over input segments that do not correspond to any top level semantic concept, it can far better recover from out of domain segments in the input, and "restart" itself on the in-domain segment that follows. However, the pre-breaking procedures that we use in conjunction with GLR* have significantly reduced this problem, since long utterances are broken into shorter sub-utterances that are parsed separately. Pre-breaking benefits Phoenix only slightly, mainly due to some better resolution of time expression attachment ambiguities. At the current time, Phoenix uses only very simple disambiguation heuristics, and does not have a mechanism similar to the parse quality heuristic of GLR*, which allows the parser to self-assess the quality of the produced result.

Because, each of the two parsing architectures appears to perform better on different types of utterances, they may hopefully be combined in a way that takes advantage of the strengths of each of them. One strategy that we have investigated is to use Phoenix as a back-up parser to GLR*. The parse result of GLR* is used whenever it is judged by the parse quality heuristic to be "Good". Whenever the parse result from GLR* is judged as "Bad", the translation is generated from the corresponding output of the Phoenix parser. Initial test evaluations using this strategy have come up with mixed results. Our conclusion has been that such combination schemes need to be further investigated.

6.6. GLR* Performance on ATIS Domain

In this section we analyze a performance evaluation of the GLR* parser on data in the ATIS domain. Whereas the analysis grammars we use in the scheduling domain of the JANUS project are semantic grammars, the grammar developed for the ATIS domain is syntactic. The evaluation was performed on top-best hypotheses of the speech recognizer. The ATIS setup of the GLR* parser attempts to recover from recognition substitution errors by using a simple word confusion matrix. We evaluate the robustness and accuracy of GLR* in this domain, by comparing it with a similar evaluation using GLR, the original non-robust version of the parser.

	GLR		GLR*		Difference	
	number	percent	number	percent	number	percent
Total Test Set	120	100.0%	120	100.0%	-	-
Parsable	62	51.7%	119	99.2%	57	47.5%
Unparsable	58	48.3%	1	0.8%	57	47.5%
Acceptable Parses	60	50.0%	73	60.8%	13	10.8%
Marked Good w/Acceptable Parse	60	50.0%	71	59.2%	11	9.2%

Table 6.13 Robustness Results: GLR vs. GLR* on ATIS Speech Input

	GLR		GLR*	
	number	percent	number	percent
Total Parsable Marked Good	62	100.0%	96	100.0%
Marked Good w/Acceptable Parse	60	96.8%	71	74.0%
Parsable Marked Bad	0	-	23	100.0%
Marked Bad w/Bad Parse	0	-	21	91.3%

Table 6.14 Accuracy Results: GLR vs. GLR* on ATIS Speech Input

The details of the grammar developed for the ATIS domain were previously described in section 4.6.1. A set of 300 sentences were used for grammar development and 250 of these sentences were used for training the parser disambiguation statistics. A list of common appearing substitutions was constructed from the development set. The parameters of the parse evaluation heuristics and the parse quality heuristic were set in the way described in section 5.5.2.

For evaluation, we used a set of 120 unseen speech recognized sentences. This is the same test set that was used in the evaluations described in section 4.6.1 and section 5.5.2. Parse results were evaluated by hand to determine whether the selected parse was acceptable.

The results comparing GLR and GLR* on ATIS were analyzed in a similar fashion to the JANUS evaluations. Table 6.13 looks at robustness by comparing the parsing coverage of both parsers. Table 6.14 compares the parsing accuracy of GLR and GLR*. Once again, the robustness results indicate a substantial gain in coverage when using GLR*. The portion of parsable sentences increased from 51.7% for GLR to 99.2% for GLR*. However, the portion of parsable sentences with an acceptable parse increased only by 10.8% (from 50.0% for GLR, to 60.8% for GLR*). The accuracy results in Table 6.14 were already presented in Table 5.4. The results indicate that while the portion of bad parses out of the parses marked "Bad" by the parser is still very high (91.3%),

the parse quality heuristic is somewhat less effective on parses marked "Good". Only 74% of the parses marked "Good" were in fact acceptable. Further inspection, however, indicated that 11 marked "Good" sentences had bad parses due to incorrect ambiguity resolution. If the correct ambiguity were chosen for these parses, the portion of good parses out of parses marked "Good" would reach 85%, similar to the comparable figures found in the JANUS evaluations.

Chapter 7

Parsing Speech Lattices using GLR*

7.1. Introduction

In the previous chapter, we investigated the performance of GLR* on parsing spontaneously spoken utterances, on both transcribed input as well as on best scoring hypotheses produced by our speech recognition system. The motivation for processing transcribed speech is that it allows us to separate the difficulties inherent in parsing spontaneous speech from the types of errors that are introduced in the speech recognition process. Transcribed input can be viewed as "optimal" speech recognition, and the parsing performance achieved on transcribed input can thus be seen as an upper-bound on the expected performance of the system when processing actual speech input.

However, in practice, the top-best hypotheses produced by our speech recognition system are far from perfect. Speech recognition errors hinder the ability of the parser to find a correct analysis for the utterance, and the effects of this are reflected in the disparity between our performance results on transcribed and speech recognized input. Whereas with transcribed input, the JANUS system using GLR* produced an acceptable end-to-end translation about 85% of the time on an unseen English test set, the comparable performance with speech recognized input on the same test set was only 46%. In fact, a speech recognition error was identified as the primary cause of a parsing failure in about 87% of all bad parses. A separate analysis of the performance of the speech recognizer on this test set revealed that the word accuracy rate was about 65%, and of the 99 utterances in the test set, only 4 (rather short) utterances were completely free of recognition errors.

The speech recognition process produces multiple hypotheses for a given speech utterance, which can be most efficiently represented in the form of a word lattice. Each hypothesis corresponds to a path of connecting words through the lattice. It is often the case that a more accurate recognition path exists through a lattice, even though it is not the best scoring hypothesis in the lattice. Processing multiple speech hypotheses instead of a single hypothesis thus has the potential of detecting a hypothesis with fewer recognition errors, which should lead to an improvement in the overall translation performance.

In this chapter, we investigate how GLR* can be adapted into a parser capable of directly parsing speech produced word lattices. We begin with a deeper analysis of the possible interface methods between the speech recognition system and the language processing components, and emphasize the potential advantages of using a lattice processing parser. We then describe the modifications to the GLR* parsing algorithm that are necessary in order to adapt it to parsing word lattices. Next, we describe how GLR* is incorporated into a version of the JANUS system that processes word lattices. This is followed by an example of how the lattice processing system works. Finally, we present a preliminary evaluation of the lattice processing version of the JANUS system.

7.2. The Advantages of Parsing Speech Lattices

In the design of a system like JANUS, choices have to be made about how best to combine the speech recognizer with the language processing. The most simple interface is the one we have seen in the previous chapter, where the speech recognition system, operating independently, chooses a single hypothesis that it deems best, and then passes this hypothesis to the parser for analysis. The main drawback of this method is that the accuracy of the top-best speech hypothesis puts an a-priori limit on the accuracy of the analyses produced by the parser. Errors introduced in the speech recognition stage cannot be corrected in the language analysis stage. Furthermore, only limited linguistic knowledge (a bigram or trigram language model) is applied in the course of speech recognition. The much stronger grammatical constraints are applied only in the course of parsing, after the "best" speech hypothesis has already been selected. Tighter integration between the speech recognizer and the parser has the potential of applying the linguistic knowledge encoded in the grammar into the speech recognition process, in order to improve the end-to-end performance of the system, as well as the quality of the speech recognition process itself. The basic premise behind such an approach is that applying the linguistic constraints encoded by the grammar on a set of possible speech hypotheses can help determine a more accurate hypothesis (or possibly a hypothesis with less "important" errors).

Tighter integration between the speech recognizer and the parser has received growing attention in the last few years. We have already mentioned some of the more notable works on this problem in section 1.3.2. An overview of the most common approaches to this problem can be found in [29] and [102]. The following are the four major conceptually different ways in which such a tighter integration between the speech recognition and grammatical constraints can be achieved:

1. **Incorporate grammar constraints directly into the speech search:**
   This is perhaps the most direct and attractive approach, but turns out to be technically very difficult. In particular, it is unclear how flexible grammatical constraints, such as those resulting from the skipping capabilities of GLR*, can be efficiently integrated directly into the speech recognition's search algorithm. Nevertheless, there has been some recent work on integrating grammatical constraints into the speech search using standard parsing techniques. Hauenstein and Weber [30], for example, investigate several ways in which this can be done.

2. **Attempt to correct the top-best hypothesis of the recognizer:**
   The idea here is to use the grammar during parse time to try and correct speech recognition errors, by inserting, deleting or substituting words from the speech hypothesis. GLR* can in fact be viewed as partially following this approach. The word skipping capabilities of GLR* allow it to delete words from the speech hypothesis. GLR* can also handle substitutions by using a confusion matrix. The matrix lists possible alternatives for words that are frequently misrecognized by the speech recognizer, and the parser attempts to substitute the speech recognized word with each of its alternatives (incurring a penalty for the substitution), in search for the most grammatical option. Tests we conducted using GLR* with a substitution matrix on ATIS data showed a slight improvement in performance. Attempting to correct words that were *deleted* from the speech hypothesis by allowing the parser to consider insertions appears to be infeasible in the context of large real-world grammars covering a reasonably large domain, as is the case in the JANUS project. It is therefore difficult to

develop mechanisms that can effectively use the constraints of the grammar to post correct speech recognition errors.

3. **Parse the n-best hypotheses produced by the speech recognizer:**
   This approach is technically simple, but inefficient. In this case, we extract a ranked list of the n best hypotheses found by the speech recognizer and use the parser to parse each of them. If we wish to require that the hypothesis be completely grammatical, the parser can be used as a filter, ruling out ungrammatical hypotheses, so that the best scoring hypothesis that is also grammatical can be selected. However, using the more flexible grammaticality constraints of GLR* should allow us to successfully parse most of the hypotheses. The score of the best parse found for each hypothesis should reflect the grammatical preferences between the various hypotheses. The parse score can then be combined with the score from the speech recognizer, in order to re-rank the list of parse results, so that the overall "best" one is ranked first.

   The main drawback of this method is its inefficiency. In order to significantly improve the accuracy of the system, a large number of hypotheses must be considered. However, parsing an n-best list of hypotheses increases the overall parsing time by a factor of n. Thus, the number of hypotheses that can be considered is limited, if the system is to perform within reasonable bounds on time and space. Furthermore, although many of the different hypotheses on the n-best list differ only slightly, each of the hypotheses is parsed in its entirety. Thus, segments that are common to many different hypotheses are parsed multiple times.

4. **Parse the speech lattice directly:**
   Parsing the speech lattice directly attempts to efficiently accomplish the same results of parsing an n-best list. Instead of first extracting the list of hypotheses from the speech lattice, and then parsing each of them separately, we use a version of the parser that can parse the lattice directly. Each complete path of connecting words through the lattice corresponds to a hypothesis. The speech score of a hypothesis is the sum of the scores of the words of which it consists. The top-best speech hypothesis corresponds to the path through the lattice that is minimal in its overall score. Each word in the lattice is parsed only once, although it may contribute to many different hypotheses. The lattice parser produces a large set of possible parses, corresponding to the parses of various complete word paths through the lattice. This set of parses can be scored and ranked according to the parse score, or according to some optimized combination of the parse score and recognizer score.

We chose to follow this fourth approach. Since the actual raw lattices produced by the speech recognizer are often too large and redundant to be parsed directly, we have developed a pre-processor that produces a significantly pruned lattice of a size that can be parsed within reasonable time and space. The lattice pre-processor is described in section 7.4. Parsing the pruned lattice is still generally equivalent to the parsing of a fairly large n-best list of hypotheses.

A small portion of a typical speech lattice can be seen in Figure 7.1. The nodes of the graph represent start or end points of lattice words. The words themselves are marked on the arcs of the graph. A score is associated with each of the lattice words. This score is a combination of an acoustic score which is derived from speech models used for the speech recognition, and a language

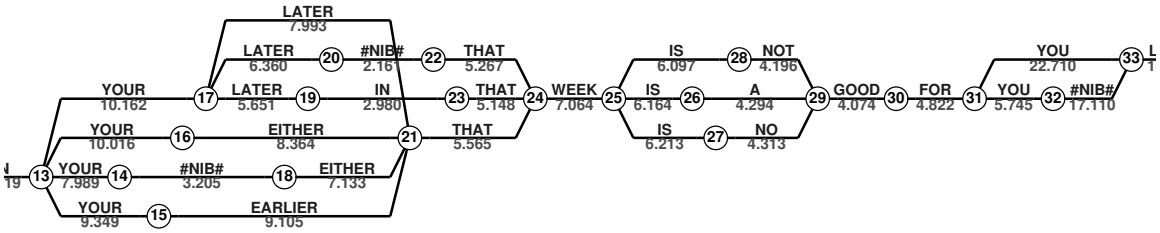


Figure 7.1 Small Portion of a Speech Lattice

```
(1)  YOUR LATER THAT WEEK IS NOT GOOD FOR YOU          (score: 72.683)
(2)  YOUR LATER THAT WEEK IS NOT GOOD FOR YOU #NIB#    (score: 72.828)
(3)  YOUR LATER THAT WEEK IS A GOOD FOR YOU            (score: 72.848)
(4)  YOUR #NIB# EITHER THAT WEEK IS NOT GOOD FOR YOU   (score: 72.855)
(5)  YOUR LATER IN THAT WEEK IS NOT GOOD FOR YOU       (score: 72.904)
(6)  YOUR EITHER THAT WEEK IS NOT GOOD FOR YOU         (score: 72.908)
(7)  YOUR LATER #NIB# THAT WEEK IS NOT GOOD FOR YOU    (score: 72.913)
(8)  YOUR LATER THAT WEEK IS NO GOOD FOR YOU           (score: 72.916)
(9)  YOUR EARLIER THAT WEEK IS NOT GOOD FOR YOU        (score: 72.982)
(10) YOUR LATER THAT WEEK IS A GOOD FOR YOU #NIB#      (score: 72.993)
```

Figure 7.2 10-best List Corresponding to the Example Speech Lattice

model probability. Each complete path of connecting words through the lattice corresponds to a hypothesis. Typical large speech lattices produced by our speech recognition system often have hundreds or even thousands of embedded hypotheses. A simple dynamic programming procedure can extract the best n hypotheses from a lattice (for any given n). Figure 7.2 shows the ranked list of the ten best hypotheses that were extracted from the small lattice in Figure 7.1. In total, 36 different hypotheses are represented in the lattice.

7.3. The GLR* Lattice Parsing Algorithm

7.3.1. Input Representation and Parsing Order

In the string parsing version of GLR*, the input to the parser is simply the linear string of words. The parser then parses the input in a left-to-right order, and the position of each word within the string can be easily inferred implicitly. However, in the lattice parsing version, each input word must be explicitly marked by its start and end points in the lattice. The input words thus have the form $[w\ (i, j)]$, where w is the actual word, i is the start point, and j is the end point. Although not essential for parsing, we also attach the acoustic score from the speech recognizer to each input word, so that this information will be easily accessible to a post-parsing procedure that combines the parse score and the acoustic score of the parse results.

For the lattice parser to function correctly, it is essential that the words of the lattice be parsed in a proper order. The start and end points of the words define a partial order between them. In essence, $[w_1(i_1, j_1)] < [w_2(i_2, j_2)]$ if there exists some path through the lattice in which $[w_1(i_1, j_1)]$ appears prior to $[w_2(i_2, j_2)]$. In such a case, there exists a path of words that connects points j_1 and i_2 in the lattice, and this can be detected by a procedure that checks the connectivity of points in the lattice. If $[w_1(i_1, j_1)] < [w_2(i_2, j_2)]$, then $[w_1(i_1, j_1)]$ must be parsed prior to $[w_2(i_2, j_2)]$. If there is no path in the lattice between j_1 and i_2 (or vice versa), then the order relation between the two lattice words is not defined, and there are no constraints as to which of the words should be parsed first. A simple procedure can therefore verify that the lattice words are properly ordered prior to parsing.

7.3.2. Modifications to the Algorithm

Before we describe the modifications that adapt the GLR* parsing algorithm to parsing speech lattices, let us first recall some important details about the GLR* parsing algorithm, that were previously described in Chapter 3. The major difference between the original GLR parsing algorithm and GLR* is in the distribution of shift actions to nodes of the GSS. GLR* accommodates skipping words of the input by allowing shift operations to be performed from *inactive* state nodes of the GSS. Shifting an input symbol from an inactive state node has the effect of skipping the words of the input that were processed after the creation of the state node and prior to the current word that is being shifted.

At each stage of the algorithm, the search heuristics control the distribution of shift actions to the state nodes of the GSS. Shift actions are first distributed to the active state nodes of the GSS. This corresponds to no additional skipped words at this stage. If the number of distributed shift actions at this point is less than the beam-width, shift actions are distributed to inactive state nodes as well. State nodes of the GSS have a *level* field, which contains the index of the last input word processed when the node was created. When distributing shift actions, inactive states are processed in decreasing level order, so that shift actions from more recent state nodes (with higher levels) that result in fewer skipped words are considered first. Shift operations are distributed to inactive state nodes in this way until the number of shifts distributed reaches the beam-width.

Each symbol node of the GSS has a *span* field containing a start and end point, which reflect the segment of the input that is covered by the symbol. The symbol nodes also maintain information about skipped words. When a reduce operation creates a new symbol node, the information about words that were skipped in between the right-hand side constituents of the grammar rule, as well as within the constituents themselves, is accumulated and attached to the new symbol node. The words that were skipped in between the right-hand side constituents are determined using the start and end points of each of the constituents.

The Distribution of Shift Actions

In the lattice parsing version of GLR*, the shift distribution step was modified to handle lattice input words. Let us consider the `DISTRIBUTE-SHIFT` step of a lattice word $[w\ (i, j)]$. Similar to the string parsing version of GLR*, shift actions must first be distributed to state nodes that do not result in any new skipped words. This is accomplished by first distributing shift actions to all

GSS state nodes of level i, since such nodes were created in the course of processing a previous lattice word of end point i, which directly connects with the current word being processed.

Once shift actions have been distributed to all state nodes of level i, if the number of shift actions distributed has not already acceded the beam-limit, state nodes of levels less than i are considered as well. Similar to the string parsing version, shift actions are distributed in an ordered fashion, where state nodes that result in fewer skipped words are considered first. This is done by first detecting all levels that correspond to words of distance one from the current word. Shift actions are then distributed to state nodes of these levels. If the total number of shift actions distributed is still below the beam-limit, we proceed to find the levels that correspond to words of distance two, and distribute shift actions to the state nodes of these levels as well. This process continues until the total number of distributed shift actions reaches the beam-limit.

Determining Word Connectivity and Lattice Paths

In the string parsing version of GLR*, the input string can be viewed as degenerate lattice that consists of a single connected path of input words $[w_i\ (i-1, i)]$, for $1 \leq i \leq n$ (where n is the length of the input). In such a string, for any i and j such that $i < j < n$, there exists a unique path from word i to word j in the string. Thus, shifting word w_j from a state node of level i has the effect of skipping a precise set of words, namely $w_{i+1} \cdots w_{j-1}$.

When dealing with a general word lattice, determining word connectivity and lattice paths is more complicated. For two arbitrary words $[w_1\ (i_1, j_1)]$ and $[w_2\ (i_2, j_2)]$ in the lattice, there could be zero, one, or multiple paths of lattice words that connect them. If no path exists between the two words, special care must be given not to shift the second word from state nodes that were created after processing the first word (since this cannot correspond to any valid word skipping). On the other hand, if $[w_1\ (i_1, j_1)]$ appears in the lattice prior to $[w_2\ (i_2, j_2)]$, and there exist multiple word paths between the two words, then shifting the second word from state nodes of level j_1 is valid, but each path between the words corresponds to a different sequence of skipped words.

The lattice parsing version of GLR* thus includes a procedure for determining the connectivity of two points in the lattice. Since the connectivity property is transitive and can be calculated recursively, a hash table is used to store the connectivity status of previously computed pairs of lattice points, so that the connectivity of a pair of lattice points will be calculated at most once.

We also developed a procedure for finding the "best" connecting path between two lattice points. Since this procedure is used to select between alternative sequences of skipped words, the "best" path is one for which the combined penalty for skipping the words is the smallest. This procedure is used in order to determine information about skipped words for the symbol nodes of the GSS. When a reduce operation creates a new symbol node, the information about words that were skipped in between the right-hand side constituents of the grammar rule is determined using the start and end points of each of the constituents. For example, assume the rule being reduced has the form $A \rightarrow B\ C$ and that the end point of the symbol node associated with B is i, while the start point of symbol node C is j. If $i \neq j$, then words were skipped between the two constituents. We use the above mentioned procedure in order to determine the sequence of skipped words between points i and j that has the smallest penalty.

Other Modifications to the Parser

Several other minor modifications to the parser should be mentioned. An additional field was added to the symbol node structure, in order to keep track of the set of lattice words that are "covered" by the symbol. This set contains all the words that were actually parsed in the sub-parse tree rooted by the symbol. When a reduce operation creates a new symbol node, the set of covered words of a new symbol is accumulated from the right-hand side constituent's symbol nodes. For example, assume the rule being reduced has the form $A \rightarrow B\ C$, then the set of covered words of the symbol associated with A is just the union of the covered words of the symbols associated with B and C. The union of the sets of skipped and covered words of a symbol corresponds to the sub-path in the lattice that was parsed in the process of creating the symbol node. Thus, at the top level, each "start" symbol node that corresponds to a complete parse will contain the information about the full path through the lattice (i.e. hypothesis) to which it corresponds.

The ambiguity packing procedure of the parser was also slightly modified. Two symbol nodes of the same grammar category can be packed by the lattice parser only if they both correspond to a parse of the exact same sub-path in the lattice. Thus, the union of the set of covered and skipped words of both symbol nodes must be the same. However, the pruning heuristic that was used in the ambiguity packing procedure of the string version of GLR* is used here as well. In other words, if two symbol nodes S_1 and S_2 are packable, but S_1 has more skipped words than S_2, then S_1 is simply discarded and the two nodes are not actually packed together.

7.4. The Lattice Processing Version of JANUS

The lattice parsing version of GLR* has been incorporated into a lattice processing version of the JANUS system. This required some changes in the configuration of the system. For each input utterance, instead of producing a top-best hypothesis or an n-best list, the speech recognition module outputs a complete scored word lattice. The lattice produced by the speech recognizer is usually far too large to be parsed in its entirety. It is therefore passed to a lattice processor, which breaks it into smaller sub-lattices and prunes each sub-lattice to a parsable size. Each of the sub-lattices is then separately parsed by the lattice parser. A post-parsing procedure then combines the results of parsing the sub-lattices into a ranked list of output results which is passed on to the discourse processor and to the generation module. The details of the lattice processor and the post-parsing processing are described in the following subsections.

7.4.1. Lattice Pre-processing

The "raw" lattices produced by the speech recognizer are most often far too large and redundant to be parsed in feasible time and space. Such lattices usually correspond to thousands of different speech hypotheses. The speech lattice thus has to be significantly pruned in order to be parsed. For this task, we use a lattice processor developed by Oren Glickman for the JANUS project [26]. The processing of the lattice is done in the following four major steps:

1. **Collapsing of Noise-words:**
   The lattice from the speech recognizer contains different kinds of noise-words, including speech modelled pauses, as well as human and non-human noises. Since no linguistic

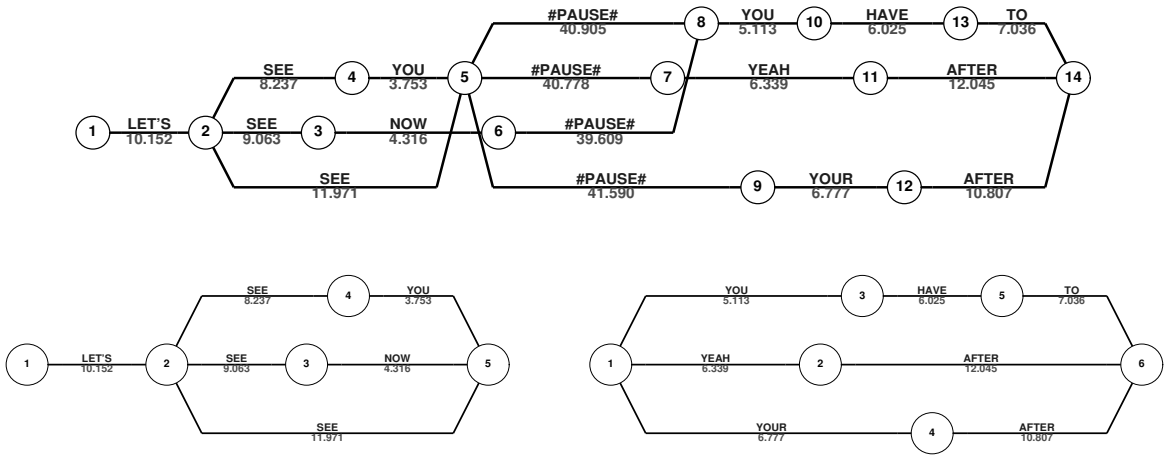


Figure 7.3 Breaking of a Speech Lattice

information is contained in these lattice words, they will be skipped and ignored by the parser. Paths in the lattice that differ only in noise-words are therefore redundant to the parser, and should be pruned. This stage of the lattice processor therefore first maps all lattice noise-words into a single generic pause. Adjacent pauses are then joined to one long pause word. We then search for sets of redundant paths in the lattice. Two paths through the lattice are redundant if they begin and end in the same nodes, and contain the same sequence of words. Only the acoustically highest scoring path of each set of redundant paths is kept, and the other paths in the set are removed from the lattice. It should be noted that no linguistic information is lost in the course of noise-word collapsing.

2. **Lattice Breaking:**
In the string parsing version of GLR*, utterances are pre-broken into smaller sub-utterances that correspond to one or several clauses. The sub-utterances are then parsed separately, significantly reducing the ambiguity facing the parser in determining clause boundaries. For similar reasons, it is useful to break the lattice of a complete utterance into sub-lattices. The lattice is broken at time intervals where only pauses and noise-words appeared in the speech signal, and where the length of the pause acceded a pre-determined threshold. The threshold is tuned to produce optimal breaking, where the emphasis is to reduce the number of false positives (breaks at places that are not clause boundaries) as much as possible. An example of a lattice before and after breaking can be seen in Figure 7.3. In the original lattice at the top of the figure, a "cut" can be made between the sub-lattice that end in nodes 5 and 6, and the sub-lattice that starts in nodes 7, 8 and 9. This is because the cut crosses only pause words, all of which have a length that is greater than the threshold. The resulting two sub-lattices appear at the bottom of the figure.

3. **Lattice Rescoring:**
Since the noise-word collapsing step condenses the lattice so that it contains less noise-words and more linguistically important information, the new lattice (or the sub-lattices after breaking) can be rescored, using either the same language model that was used for the

original lattice, or a new language model. Due to the noise-word collapsing, words that were not adjacent in the original lattice may have become adjacent in the new lattice. As a result, different bigram probabilities may effect the scores attached to the words. The rescoring may therefore change the ranking of alternative paths through the lattice. At the current time, this rescoring step has not yet been fully implemented into the lattice processor.

4. **Lattice Pruning:**
   In the last step of the lattice processing, each of the sub-lattices is pruned, to ensure that each sub-lattice is of a size that can be parsed in reasonable time and space. The pruning process removes words from the lattice one at a time in an ordered way. The intent is to remove words that contribute to bad scoring hypotheses first. For each word, a dynamic programming procedure determines the score of the best lattice path in which the word appears. The words are then pruned according to this score, in descending order. The procedure allows the user to specify the desired size of the pruned lattice, in terms of either an absolute size or a percentage of the size of the original lattice.

7.4.2. Post-parsing Processing

For each parsed sub-lattice, the parser returns a list of packed parses that were found. This list is ranked according to the score that the parser attaches to each of the alternative parse results. Parses that correspond to different lattice paths are not packed together, and thus appear separately on the list of parses found. By default, when the parser is run on a stand alone basis, the best scoring parse on the list of parses is unpacked and disambiguated, and the result is displayed to the user. However, when the parser is integrated into the lattice processing version of JANUS, the entire list of parse results is passed on to a post-parsing procedure.

The post-parsing procedure selects the top k packed parses from the list of parses (k is an adjustable constant). These parses will correspond to different paths through the lattice. Each parse is is first unpacked and disambiguated. Next, the path of lattice words associated with each of the parses is retrieved and the acoustic score of this path is calculated and attached to the parse. The acoustic score is of each parse is then combined with the parser score [1] into a joint score, and the parse list is re-ranked according to the joint scores.

After each of the individual sub-lattices are parsed and the list of parse results for each sub-lattice is rescored according to both the parser and acoustic scores, the parse results for the sub-lattices are combined together into a list of parse results for the entire utterance. Finally, the best scoring utterance parse is selected and passed on to the generation module.

We have recently also integrated the discourse processor into this version of the system. When discourse processing is performed, the list of possible utterance parses is passed on to the discourse processor. The discourse processor rates each of the possible parses with yet another score, which is combined with the parse score and the acoustic score. The list of parse results is then re-ranked according to the combination of all three scores.

[1] We are still experimenting with the weights assigned to each of the scores in this combination. In the experiments reported here the weight assigned to the parse score was three times that of the acoustic score.

```
okay {comma}
we're gonna need to meet again {comma} the next two weeks {seos}
on {comma} Thursday {comma} April first {comma}
I {comma} can meet after eleven o'clock {period}
if that's alright with you {period} {seos}
```

Figure 7.4 Transcribed Text of the Example Utterance

Speech Top-best	**Lattice**
Input Hypothesis: OKAY ARE ANY TO MEET AGAIN THE NEXT TWO WEEKS ON THURSDAY APRIL FIRST I EIGHTH CAN MEET AFTER ELEVEN O+CLOCK IF THAT+S ALRIGHT WITH YOU	**Input Hypothesis:** OKAY I COULD NEED TO MEET AGAIN THE NEXT TWO WEEKS ON THURSDAY APRIL FIRST I CAN MEET AFTER ELEVEN O+CLOCK IF THAT+S ALRIGHT WITH YOU
Parsed Clauses: (ON THURSDAY APRIL FIRST) (I EIGHTH)	**Parsed Clauses:** (I NEED TO MEET AGAIN THE NEXT TWO WEEKS) (ON THURSDAY APRIL FIRST I CAN MEET AFTER ELEVEN O+CLOCK) (IF THAT+S ALRIGHT WITH YOU)
Translated Output: "THURSDAY APRIL THE FIRST" "I THE EIGHTH"	**Translated Output:** "I HAVE TO MEET IN THE NEXT TWO WEEKS AGAIN" "ON THURSDAY APRIL THE FIRST I CAN MEET AFTER ELEVEN O+CLOCK" "IF THAT IS GOOD FOR YOU"

Figure 7.5 Example: Parsing Top-best Speech Hypothesis vs. Lattice

7.5. An Example

To demonstrate how using the lattice parser can improve both the accuracy of the selected speech hypothesis, as well as the quality of the end-to-end translation result, we present an example that was taken from a recent evaluation of the lattice parsing version of the system (see the next section for the complete results of this evaluation). This example was taken from an English-to-English translation evaluation.

Figure 7.4 shows the transcribed text of the example utterance. In Figure 7.5, the results from parsing the top-best hypothesis of the speech recognizer are compared with the corresponding results from the lattice parser. The top-best speech hypothesis is contrasted with the hypothesis (lattice path) of the parse chosen by the lattice parser. At the top of the figure, the input hypothesis can be seen. When compared with the transcribed text of the utterance, one can see that the top-

best speech hypothesis contains two misrecognized substitutions ("are", "any"), one deletion ("need") and one insertion ("eight"), which corresponds to a word accuracy of 85% for the utterance. The lattice hypothesis that was chosen as best by the parser has only two substitutions ("I", "could"), which corresponds to a word accuracy of 92%.

But most importantly, the misrecognitions in the top-best speech hypothesis had a very adverse effect on the parsability of the utterance. This resulted in a very poor parse and consequently, a poor translation result. In the lattice case, only one of the two substituted words had to be skipped in order to parse the utterance. This resulted in a generally correct parse, and consequently, a correct translation.

7.6. Performance Evaluation

Table 7.1 shows the results of a preliminary evaluation we have recently conducted to evaluate the performance of the lattice parsing approach. On a common test set of 14 unseen English dialogs (99 utterances), we compared the results of using the lattice processing version of the JANUS system with the results of processing the top-best speech hypothesis. Results on transcribed input of the same test set also appear in the table.

As can be seen in the table, the preliminary results show only a slight improvement in the total percentage of acceptable translations, which increased from 41.4% for parsing the top-best speech hypothesis, to 44.5% when parsing lattices. This compares with a performance of 83.9% acceptable translations on the transcribed input. While the percentage of acceptable translations improved slightly with using the lattice parser, the percentage of "perfect" translations decreased (10.1% for lattice parsing versus 17.2% for speech top-best). This was offset by a larger increase in the percentage of "OK" translations (34.3% for lattice parsing versus 24.2% for speech top-best). Note that the lattice parsing evaluation does not distinguish between translations that were bad due to a selected hypothesis with poor recognition, and other bad translations. The presumption was that if a mostly correct hypothesis appears in the lattice, the lattice parser should hopefully be capable of finding it.

Further analysis showed that the lattice parser succeeded to produce an acceptable translation for 15 of the utterances that were badly translated due to poor recognition when the top-best speech hypothesis was used. The hypothesis selected by the lattice parser for these utterances contained fewer misrecognitions, and consequently, the utterance was parsed and translated adequately. However, 12 utterances that were acceptably translated by using the top-best speech hypothesis, were translated poorly when using the lattice parser. In most of these cases, this was due to the fact that the lattice parser prefered a hypothesis in the lattice that produced a better parse score than the top-best speech hypothesis, but that contained more misrecognitions. Such problems could possibly be resolved by a better combination scheme between the acoustic score and the parser score, and by improvements in the parse score heuristics.

Another interesting phenomena that we have noticed is that the lattice parser appears to perform better than the speech top-best on longer utterances, while on short utterances, parsing the speech top-best hypothesis is better. We believe this is due to the fact that while the top-best speech hypothesis is likely to contain more misrecognized words for longer utterances, such utterances contain far greater amounts of "linguistic constraints", which can be captured via the relative parse scores of the various lattice paths. On the other hand, short utterances contain little linguistic

	Transcribed Input	Top-best Speech	Lattice Parser
Parsable	100.0%	100.0%	100.0%
Perfect Translation	45.5%	17.2%	10.1%
OK Translation	38.4%	24.2%	34.3%
Acceptable Translation	83.9%	41.4%	44.5%
Bad Translation	16.1%	15.2%	55.6%
Poor Recognition	-	43.4%	-

Table 7.1 Performance Results of Top-best Speech vs. Lattice Parsing on English Input (14 dialogs, 99 utterances, 07-14-95)

constraints. For example, if the lattice contains the two hypotheses "`that sounds good`" and "`that's not good`", it will be difficult for the lattice parser to distinguish which of the two hypotheses is correct, since both will result in good parse scores. In such cases, the acoustic preference should carry greater weight. We therefore plan on investigating how to better capture the level of confidence provided by each of our knowledge sources, and are hopeful that this will lead to a combination scheme with improved performance.

Chapter 8

Conclusions

8.1. Summary of Results

In this thesis, we described GLR*, a parsing system based on the GLR parsing algorithm, that was designed to be robust to speech disfluencies and limited grammar coverage. The extensive evaluations presented in the thesis demonstrate how GLR* substantially enhances the performance of the JANUS system into which it has been integrated. These evaluations provide evidence as to the suitability of the GLR* parser to parsing spontaneous spoken language.

The main results presented in the thesis are the following:

1. The unrestricted GLR* algorithm, which finds and parses *all* parsable subsets of a given input, requires infeasible time and space resources. However, feasible time and space bounds are achieved using a beam search that limits the amount of word skipping considered by the parser. Reasonably tight beam parameters were sufficient for finding the best parsable subset in the domains in which we have tested the parser.

2. We developed a statistical disambiguation module for the unification based versions of the GLR and GLR* parsers. Probabilities are attached directly to the actions in the LR parsing table. This disambiguation model is much more detailed than previous models such as probabilistic context-free grammars, and can capture some level of inter-sentential context. However, the number of parameters that need to be modeled is much larger, and the model requires training on a large set of correctly disambiguated parses. Although obtaining such a large training corpus is a tedious task, our experiments have shown that surprisingly good disambiguation results can be achieved with even relatively very small amounts of training data, when the data contains examples of the most common types of ambiguity. Our disambiguation success rate was about 66% for ambiguous sentences in the JANUS Spanish (SSST) domain, and about 75% for the ATIS domain. In both cases the training sets managed to observe only about 5% of all possible parsing actions.

3. We developed a general framework for combining a collection of parse evaluation measures into an integrated heuristic for evaluating and ranking the parses produced by the GLR* parser, and developed a tool for optimizing the weight settings of the various evaluation measures. We developed a set of four measures for the JANUS scheduling domain and the ATIS domain. The most dominant measure is a penalty function for skipped words. The other three measures are a penalty function for substituted words, a fragmentation penalty, and a statistically based penalty. The results of test evaluations showed that the combined heuristic based on these four measures is about 90% effective in selecting the best parse from the set of parses produced by the parser on a given input.

4. We developed a parse quality heuristic that allows the parser to self-judge the quality of the parse chosen as best, and to detect cases in which the best result of the parser is likely to have skipped information that is crucial to the correct understanding of the input utterance. The results of test evaluations of the parse quality heuristic show that it is about 85% accurate in predicting whether or not the selected parse analysis is reasonably correct.

5. We integrated the GLR* parser into the JANUS speech translation system, resolved integration issues such as full utterance parsing, and conducted extensive evaluations of the performance of the system using GLR*. We compared system performance using GLR* with corresponding versions that used both the original non-robust GLR parser and the semantic-based Phoenix parser. Our evaluation results on both transcribed and speech recognized input show that compared with GLR, GLR* produces between 15% and 30% more parse analyses that result in acceptable translations (on ATIS the increase was more modest - about 11%). The overall end-to-end translation performance of GLR* and Phoenix was very similar, although Phoenix appeared to be slightly better on speech recognized input.

6. We developed a version of GLR* that is suitable to parsing word lattices produced by the speech recognizer, and have begun investigating how parsing lattices can potentially overcome errors of the speech recognizer and improve end-to-end performance of the speech translation system. A preliminary test evaluation showed a very slight increase of about 3% in the number of acceptable translations produced by the system, when the system processed lattices instead of the top-best hypothesis from the speech recognizer.

8.2. Further Research

GLR* is now an integral part of the JANUS system, an evolving speech-to-speech translation project. The goal of improving the end-to-end performance of the JANUS system, and the need to address practical problems that are manifested in the actual data processed by the system, provide the impetus for much of our current and future research on improving the parser. In this last section of the thesis, we mention some immediate questions about how the parser can be improved within the current system, and to what degree it is applicable to other systems and domains.

Other Applications

As the parsing component of the JANUS speech-to-speech translation system, GLR* has proven to be highly suitable for handling spontaneously spoken language in limited domains. Evaluations on the ATIS domain have shown similar results. However, several questions remain with respect to the generality of the GLR* approach:

1. Is GLR* similarly effective for other spoken language applications?

2. Are the principles of the GLR* approach applicable to other types of parsers?

3. The search and parse evaluation heuristics that are an integral part of the parser are essential for its performance. Are these heuristics suitable for other domains as well?

4. Is GLR* useful for other practical natural language systems that do not process speech? For example, text extraction systems require an ability to scan large amounts of text, analyze the portions that are relevant to the topic of interest, and ignore the rest of the text. Can GLR*, or its principle ideas, offer a new solution to this problem?

Improved Beam Search

The current beam search employed by the parser allows only a constant number of different shift actions to be considered at each parsing stage. Although this provides some flexibility with respect to the amount of word skipping that is considered at each stage, some problems remain. Ambiguity tends to increase with the length of the input utterance, along with the number of parsing options considered by the parser. Thus, significantly more shift actions can be distributed in later stages, and since the size of the beam remains constant, this has an effect of reducing the amount of word skipping allowed. This problem is even more severe in the lattice parsing version of the algorithm, where even a much greater number of possibilities are considered simultaneously. We must therefore investigate how to implement a more flexible beam mechanism, and possibly consider pruning parsing actions according to probabilistic or other criteria.

Improved Parse Evaluation and Parse Quality Heuristics

The parse evaluation heuristic that we developed for the JANUS and ATIS grammars combines four parse scoring measures. Our evaluations indicated that this combined heuristic was quite effective in selecting the best parse among possible parses of different subsets of the input utterance. Nevertheless, adding additional scoring measures should potentially further improve the heuristic. We are currently investigating how the discourse processor can apply a variety of discourse constraints, and how the compliance with these constraints can be expressed in the form of penalty scores. We are also looking into the issue of how the discourse scores can be combined with the parser's other scoring measures in the best possible way, and are investigating schemes that are more flexible than linear combination.

The parse quality heuristic that determines the confidence of the parser in the quality of the best parse found can also be improved. Deeper semantic and pragmatic analysis should allow us to better judge whether the information that was skipped was crucial to the understanding of the utterance. Furthermore, we would like to try and expand this heuristic into a confidence heuristic for the full end-to-end speech translation system. In such a case, the heuristic would have to capture measures of confidence from the various components of the system, most notably the speech recognizer, the parser and the discourse processor.

Combining GLR* and Phoenix

Because the architectures of the GLR* parser and the Phoenix parser are very different, they each perform well for different kinds of utterances. We would like to further investigate these differences, in order to hopefully develop a strategy for combining the two parsers in an advantageous way. One direction that we are investigating is using Phoenix as a back-up parser. In this configuration, the input is first parsed by GLR*, but if all resulting parses are considered bad by the parse quality heuristic, the input is reparsed by the Phoenix parser. We are in the process of investigating whether this combination method can improve end-to-end translation results.

Statistical Disambiguation

Statistical disambiguation is currently performed as a post process to parsing. As explained in section 4.5.1, due to the non local effects of feature structure unification, it is difficult to use probabilistic information at parse time in order to prune out sub-parses with low probabilities. Nevertheless, this is an attractive direction that we would like to explore further, since an improved search heuristic that includes such pruning can substantially improve parser efficiency without the cost of reduced accuracy. This is especially important for the lattice parsing version of GLR*, where the parser must consider an extremely large number of possibilities.

Another problem is the amount of training data available for calculating the statistics used for disambiguation. For both the JANUS and ATIS grammars, the size of the training set was inadequately small. Extending the training corpus of sentences and their correct feature structures with large amounts of additional data is likely to significantly improve disambiguation. However, is it possible to do this with far less manual effort, or even automatically? Currently, this is still a rather labor intensive task, since the correct feature structure must either be hand-coded (as a target ILT), or selected from a set of ambiguities using our interactive disambiguation procedure. One possibility worth exploring is to use manually graded data from our frequent end-to-end evaluations to automatically determine "correct" feature structures. If "perfect" translations can indicate with very high confidence that the ILT feature structure of the selected parse was completely correct, then sentences that were "perfectly" translated can subsequently be automatically added to the statistical training corpus, along with their chosen feature structure. This idea remains to be investigated.

The probabilistic model that we currently use for statistical disambiguation does not take into account which words of the utterance were skipped by the parser. Thus, word skipping is not modeled probabilistically, and is handled by a separate penalty scheme. We would like to investigate whether the probability of skipping a word can be captured by an improved probabilistic model, and whether this can improve both statistical disambiguation and parse selection.

Lattice Parsing

The preliminary results on parsing lattices did not show a significant improvement in end-to-end translation accuracy, compared to results from parsing the top-best speech recognizer. This was partly due to the fact that the lattices had to be severely pruned, and that in some cases, the most accurate hypothesis in the lattice was pruned out, while the hypotheses that remained in the lattice were not significantly better than the top-best one. If the efficiency of the lattice parser can be significantly improved, less pruning will be required in advance, since we will be able to parse larger lattices. This is likely to improve our performance on lattice parsing.

Better schemes for combining the parse score, acoustic score and possibly other measures (such as discourse constraints) must also be investigated. As mentioned, we noticed that the lattice parser appears to perform better than the speech top-best on longer utterances, while on short utterances, parsing the speech top-best hypothesis is better. We believe this is due to the fact that longer utterances contain far greater amounts of "linguistic constraints", which are reflected via the parse scores. We therefore plan on investigating how to better capture the level of confidence provided by each of our knowledge sources, and are hopeful that this will lead to a combination scheme with improved performance.

Appendix A

Lisp Code of the GLR* Parsing Algorithm

This Appendix contains a detailed description of the GLR* parsing algorithm, presented in the form of a package of data-structures and functions written in Common Lisp. This package of lisp code is a simplified version of the actual implementation of GLR* within the unification-based Generalized LR Parser/Compiler software package [96]. In particular, all unification related operations have been deleted. Thus, the code presented here describes only the context-free aspects of the basic GLR* parsing algorithm (including the search heuristics). The full Lucid Common Lisp implementation package of GLR* and the unification-based Generalized LR Parser/Compiler is available for academic research from the Center for Machine Translation at Carnegie Mellon University, subject to a licensing agreement. Interested parties should contact the author.

A.1. Global Variables

```
;;;*****************************************************************************
;;; Parser Global Variable Settings
;;;*****************************************************************************

(defparameter *accept* 'A)          ; 'a
(defparameter *shift* 'S)          ; 's
(defparameter *wild-card-shift* 'SH*)
(defparameter *reduce* 'R)          ; 'r
(defparameter *wild-card-character* '%)
(defvar *no-ambiguity-packing* nil)     ; If you do not want to pack
                                        ; ambiguities, set this t.

;; parsing tables variables

(defvar *grammar-table*)   ; array of grammar rules
(defvar *action-table*) ; array of action tables
(defvar *goto-table*) ; array of goto tables
(defvar *reduce-union-table*)   ; for recovery from failure

(defvar *GG*) ; rules, action table and goto table are bound
(defvar *A-TAB*) ; to these variables when the grammar is
(defvar *G-TAB*) ; loaded.

(defvar *start-symbol*) ; rhs of the first rule.
```

```
;; Graph Structured Stack variables

(defvar *state-array*) ; array for state vertex
(defvar *symbol-array*) ; array for symbol vertex
(defvar *node-array*) ; array for node

(defparameter *initial-vertex-size* 1000)
(defparameter *initial-node-size* 1000)
(defparameter *initial-state* 0 "initial state number")


;;; Additional global variables for GLR*

(defvar *input-list*)           ; assoc list of input words keyed by positions
(defvar *parsed-sent*)          ; list of sentence words actually parsed
(defvar *skipped-words*)        ; list of words that were skipped
(defvar *subst-words*)          ; list of words that were substituted
(defvar *skip-word-limit* nil)  ; AL - this limits the maximum number of words
                                ; in a row allowed to be skipped
                                ; nil means unlimited
                                ; 1 means NO SKIPPING
                                ; and it may be set to any integer k
(defvar *beam-limit* 5)         ; AL - this limits the maximum number of top
                                ; nodes to be pursued
                                ; nil means unlimited
                                ; and it may be set to any integer k
(defvar *saved-active-array*)   ; array of previous active nodes - AL
(defvar *speech-subst-list* nil) ; AL - this assoc list contains typical
                                 ; substitutions of the speech recognizer
                                 ; should be loaded separately for each task
(defvar *disambig-flag* nil)    ; AL - turn this flag on for statistical
                                ; disambiguation. nil means no disambiguation
(defvar *last-lhs* nil)         ; AL - keeps track of last reduced constituent
```

A.2. Data Structures

```
;;;*****************************************************************************
;;; Parser Structure Definitions
;;;*****************************************************************************

; **** structure for active-list

(defstruct active-vertex
    vertex-number ; state node pointer
    level                       ; word end position of last word shifted - AL
    state ; state number
    action ; list of actions e.g. ((re . 3)(sh . 5))
    )
```

```
; **** structure for state-vertex

(defstruct (state-vertex (:conc-name nil))
    state ; state number  e.g. 4
    level                       ; word-position of last word shifted - AL
    par-symbol ; e.g. (3 5)
    )

; **** structure for symbol-vertex

(defstruct (symbol-vertex (:conc-name nil))
    category            ; e.g. NP
    node-ptr         ; e.g. 3
    par-state         ; e.g. (3 5)
    start-position      ; e.g. 13 start word position of the left most symbol
                        ;         add 5/26/87 to pack every packable ambiguity
    skip                ; list skipped words in the phrase - AL
    subst-list          ; list of substituted words in the phrase - AL
    )


;;;*****************************************************************************
;;; Initialization and structure modification functions
;;;*****************************************************************************

;;; init-array initializes the arrays of the three main structure types

(defun init-array ()
    (setq *state-array* (make-array *initial-vertex-size*
                            :adjustable t
                            :fill-pointer t))
    (setf (fill-pointer *state-array*) 0)
    (setq *symbol-array* (make-array *initial-vertex-size*
                            :adjustable t
                            :fill-pointer t))
    (setf (fill-pointer *symbol-array*) 0)
)

; state-array
(defun state-array (n)
    (aref *state-array* n))

; symbol-array
(defun symbol-array (n)
    (aref *symbol-array* n))

;;; push-state adds a new state vertex to the state array

(defun push-state (state-number level par-symbol)
    (let ((a-vertex (make-state-vertex :state state-number
                                       :level level                 ; AL
                                       :par-symbol par-symbol)))
          (vector-push-extend a-vertex *state-array*)  ; return value
```

```
    ))

;;; push-symbol adds a new symbol vertex to the symbol array

(defun push-symbol (category node-ptr par-state-list start-position
                    skip-list subst-list)
    (let ((a-symbol (make-symbol-vertex :category category
                        :node-ptr node-ptr :par-state par-state-list
                        :start-position start-position
                        :skip skip-list                          ; AL
                        :subst-list subst-list)))                ; AL
         (vector-push-extend a-symbol *symbol-array*)  ; return value
    ))

;;; push-node adds a parse node to the node array

(defun push-node (category sons value)
    (let ((a-node (make-node :node-category category
                             :sons sons :value value))
          node-no)
         (setq node-no (vector-push-extend a-node *node-array*))
         node-no
    ))

;;; push-symbol-node:  push a symbol and a node

(defun push-symbol-node (category par-state sons value word-position
                         skip-list subst-list)
    (push-symbol category (push-node category sons value)
        par-state word-position skip-list subst-list))         ; AL

;;; add-par-state adds a parent state to list of parent states of a symbol

(defun add-par-state (parent symbol)
    (setf (par-state (aref *symbol-array* symbol))
          (append (par-state (aref *symbol-array* symbol))
                  (list parent))))

;;; add-par-symbol adds a parent symbol to list of parent symbols of a state

(defun add-par-symbol (parent state)
    (setf (par-symbol (aref *state-array* state))
          (append (par-symbol (aref *state-array* state))
                  (list parent))))

;;; delete-par-symbol deletes a parent symbol from the list in a state vertex

(defun delete-par-symbol (a-parent state)
    (setf (par-symbol (aref *state-array* state))
          (remove a-parent (par-symbol (state-array state)))
    ))
```

A.3. Main Parsing Functions

```
;;;*****************************************************************************
;;; main parser lisp code
;;;*****************************************************************************

;;; parse-list is the main GLR* parsing function for word lists (input strings)

(defun parse-list (text)
    (setq *parser-failed* nil)
    (setq *input-list* nil)
    (setq *parsed-sent* nil)
    (setq *skipped-words* nil)
    (setq *subst-words* nil)
    (setq *saved-active-array*
       (make-array (length text) :initial-element nil))
    (do ((active-list (initialize))
         (active-level)
         (rest-of-text text (cdr rest-of-text))
         (lookahead)
         (parsed-part nil) ; display when failed
         (shift-active-list nil)
         (start-time (get-internal-real-time))
         (time-spent)
         (word-position 1 (1+ word-position))
         (active-before-reduce) ; active at the beginning of the session
        )
        ((null rest-of-text) nil) ; return
        (setq lookahead (car rest-of-text))
        (push (list word-position lookahead) *input-list* )

; AL - check for noise word, if noise don't parse

        (when (not (is-noise lookahead))

; AL - append current list of active nodes to the saved-active-array

           (setq active-level (active-vertex-level (car active-list)))
           (setf (aref *saved-active-array* active-level)
              (append (copy-active-list active-list)
                      (aref *saved-active-array* active-level)))

           (setq active-list
              (fill-actions active-list lookahead))  ; of active list

           (cond ((and (not *ignore-space*)
                       (punctuation-p lookahead))
                  (when (eq *space-character* lookahead)
                     (setq rest-of-text (cdr rest-of-text))
                     (setq lookahead (car rest-of-text)))
                  (setq active-list (reduce-all active-list lookahead)))
                 (*ignore-space*
                      (setq active-list (reduce-all active-list lookahead)))
                 (t (setq active-list (remove-action active-list *reduce*))))
```

```
; AL - merge states on *saved-active-array* and then fill in shift actions

           (setf (aref *saved-active-array* active-level)
                 (merge-state (aref *saved-active-array* active-level)))

           (multiple-value-setq (active-list shift-active-list)
              (fill-shift-actions lookahead))

; AL - failure occurs only at the end of the sentence

           (when (and (eq lookahead '$)
                      (null active-list))
              (setq *parser-failed* t)
              (setq lookahead (car rest-of-text)))

; AL - Exit at end-of-string: list of found parses is in active-list

           (when (eq lookahead '$)
                 (return-from parse-list         ;; ****** exit
                    (disp-parses active-list time-spent)))

; AL - do all the shifts

           (setq active-list (shift-all shift-active-list word-position))

; AL - merge top states again

           (setq active-list (merge-state active-list))

           (setq parsed-part (cons lookahead parsed-part)))))


;;; initialize starts the structure arrays at run time

(defun initialize ()
    (setf (fill-pointer *state-array*) 0)
    (setf (fill-pointer *symbol-array*) 0)
    (setf (fill-pointer *node-array*) 0)
    (list (make-active-vertex
             :vertex-number (push-state *initial-state* 0 nil)     ; AL
             :level 0                                              ; AL
             :state *initial-state*
             :action nil)))


;;; fill-actions fills the parser actions defined for an active node

(defun fill-actions (active-list lookahead)
   (let (action (return-active active-list))
      (dolist (an-active active-list return-active)
         (setq action (get-action (active-vertex-state an-active) lookahead))
         (cond ((null action)
                (setq return-active
```

```
                    (remove-from an-active return-active))
                (t (setf (active-vertex-action an-active) action)))))))


;;; AL - fill-shift-actions fills actions and then prunes to leave only shifts
;;;      also prunes according to the setting of *skip-word-limit* and
;;;      *beam-limit*

(defun fill-shift-actions (lookahead)
   (let ((filled-active nil)
         (filled-active-list nil)
         (shift-action-list nil)
         (active-list nil)
         (return-active nil)
         (start-return-active nil)
         (shift-return-active nil)
         (substitution-list nil)
         (top-level (caar *input-list*))
         (first-level (- (caar *input-list*) 1))
         (cur-level nil)
         (node-level nil))

      (setq substitution-list
         (cons lookahead (second (assoc lookahead *speech-subst-list*))))

;;; first distribute shifts to the initial state to allow parse to start here

      (setq active-list (copy-active-list (aref *saved-active-array* 0)))

      (when (not (null *beam-limit*))
         (dolist (an-active active-list nil)
            (dolist (a-look substitution-list filled-active-list)
               (setq filled-active
                  (car (fill-actions (list an-active) a-look)))
               (when filled-active
                  (setq shift-action-list
                  (shift-prune (active-vertex-action filled-active)))
                  (when shift-action-list
                     (setf (active-vertex-action filled-active)
                           shift-action-list)
                     (setq start-return-active
                        (cons (list filled-active a-look)
                              start-return-active)))))))

;;; outer loop is on the distance from the top level

         (do ((distance 0 (+ distance 1)))
             ((> distance first-level)
                 (setq shift-return-active
                    (append shift-return-active start-return-active))
                 (values (mapcar #'car shift-return-active)
                         shift-return-active))

           (setq active-list nil)
```

```
(setq cur-level (- first-level distance))

(when (or (> cur-level 0)
          (null *beam-limit*))
   (setq active-list
      (copy-active-list (aref *saved-active-array* cur-level))))

(dolist (an-active active-list nil)
   (setq filled-active-list nil)
   (dolist (a-look substitution-list filled-active-list)
      (setq filled-active
         (car (fill-actions (list (copy-one an-active)) a-look)))
      (when (not (null filled-active))
         (setq filled-active-list
            (append filled-active-list
               (list (list filled-active a-look))))))

   (dolist (filled-active filled-active-list nil)
      (setq node-level
         (level (state-array
            (active-vertex-vertex-number an-active))))
      (setq shift-action-list
         (shift-prune (active-vertex-action (first filled-active))))
      (when (null shift-return-active)
         (setq first-level node-level))
      (cond ((null shift-action-list)
                (setq *active-killed*
                   (cons filled-active *active-killed*)))
            ((and (not (null *skip-word-limit*))
                  (< node-level (- top-level *skip-word-limit*)))
                (setq *active-killed*
                   (cons filled-active *active-killed*))
                (setq shift-return-active
                   (append shift-return-active start-return-active))
                (return-from fill-shift-actions
                   (values (mapcar #'car shift-return-active)
                           shift-return-active)))
            ((and (not (null *beam-limit*))
                  (< node-level first-level)
                  (> (length return-active) *beam-limit*))
                (setq *active-killed*
                   (cons filled-active *active-killed*))
                (setq shift-return-active
                   (append shift-return-active start-return-active))
                (return-from fill-shift-actions
                   (values (mapcar #'car shift-return-active)
                           shift-return-active)))
            (t (setf (active-vertex-action (first filled-active))
                     shift-action-list)
               (setq return-active
                  (cons (first filled-active) return-active))
               (setq shift-return-active
                  (cons filled-active shift-return-active)))))))
```

```
      (values (mapcar #'car shift-return-active) shift-return-active)))


;;; AL - shift-prune prunes a list of actions and leaves only shifts

(defun shift-prune (action-list)
   (let ((return-actions nil))
      (dolist (an-action action-list return-actions)
         (when (not (eq *reduce* (first an-action)))
            (setq return-actions (cons an-action return-actions))))))


;;; pickup-reduce   pick up a reduce action from the active vertex list
;;;                and also returns the rest of the active list.
;;; e.g. ((1 5 ((sh . 4) (re . 3))) (2 10 (re . 4)))
;;;   --> (1 5 (re . 3)) and ((1 5 ((sh . 4))) (2 10 (re . 4)))

(defun pickup-reduce (active-list)
    ; first find the "right most reduce"
   (let (the-reduce the-active (max-word-position 0)
         new-actions return-active copy)
      (do ((rest-active active-list (cdr rest-active))
   (an-active))
  ((null rest-active) nil)
         (setq an-active (car rest-active))
         (do ((rest (active-vertex-action an-active) (cdr rest))
              (an-action) (word-position) (rule))
             ((null rest) nil)
            (setq an-action (car rest))
            (when (eq *reduce* (first an-action))  ; a reduce found
               (setq rule (get-rule (cdr an-action)))
               (setq word-position
                  (get-right-most (active-vertex-vertex-number an-active)
                                  (second rule)))                          ; AL

               (when (or (> word-position max-word-position)
                         (and (= word-position max-word-position)
                              (equal (first rule) *last-lhs*)))
                  (setq max-word-position word-position)
                  (setq the-reduce an-action)
                  (setq the-active an-active)))))

      (cond ((= max-word-position 0) (values nil active-list))
            (t (setq new-actions
                  (remove-from the-reduce
                     (active-vertex-action the-active)))
               (cond ((null new-actions)
                      (setq return-active
                         (remove-from the-active active-list)))
                     (t (setf (active-vertex-action the-active) new-actions)
                        (setq return-active active-list)))
               (setq copy  (copy-active-vertex the-active))
               (setf (active-vertex-action copy) (list the-reduce))
               (values copy return-active))))))
```

```
;;; get-right-most
;;; AL - rewritten to correctly go all the way back to first constituent
;;;      and find its start position

(defun get-right-most (vertex-number rule-right)
   (do ((rest (par-symbol (state-array vertex-number)) (cdr rest))
        (a-symbol)
        (new-start)
        (start-pos 0))
       ((null rest) start-pos)
      (setq a-symbol (car rest))
      (setq new-start (get-right-symbol a-symbol rule-right))
      (if (> new-start start-pos)
          (setq start-pos new-start))))


;;; get-right-symbol
;;; AL- returns start position if first symbol in right-rule, else goes back
;;;     further to parent state

(defun get-right-symbol (symbol-number rest-right-rule)
   (if (= (length rest-right-rule) 1)
       (first (start-position (symbol-array symbol-number)))
       (do ((rest (par-state (symbol-array symbol-number)) (cdr rest)) ; else
            (a-state)
            (new-start)
            (start-pos 0))
           ((null rest) start-pos)
           (setq a-state (car rest))
           (setq new-start (get-right-most a-state (cdr rest-right-rule)))
           (if (> new-start start-pos)
               (setq start-pos new-start)))))


;;; merge-state
;;;  Merge right-most states that have the same state numbers.
;;;  Returns the new (merged) active list.
;;;  The merged one will be removed from he active list.
;;;  The variable pairs is an assoc list of (state-number vertex-number) pair.
;;;  called after all the reduce actions are completed and after shifts.

(defun merge-state (active-list)
   (do ((rest active-list (cdr rest))
        (an-active) (return-active nil)
        (cur-state)
        (cur-vertex-no)
        (pairs nil)       ;   assoc list of (state vertex-numbers)
        (the-pair))
       ((null rest) (reverse return-active))  ; *** return
      (setq an-active (car rest))
      (setq cur-state (active-vertex-state an-active))               ; AL
      (setq cur-vertex-no (active-vertex-vertex-number an-active))
```

```
      (cond ((setq the-pair (assoc cur-state pairs))

; if the active-vertex has the same state number as the-pair
             (do ((prev-vertexes (cadr the-pair) (cdr prev-vertexes))
                  (a-prev))
                 ((null prev-vertexes)
                (setq pairs
                   (cons (list cur-state
                               (cons cur-vertex-no (cadr the-pair)))
                         pairs))
                (setq return-active (cons an-active return-active)))
                (setq a-prev (car prev-vertexes))
                (when (= (level (state-array cur-vertex-no))
                         (level (state-array a-prev)))
                   (remove-from an-active active-list)

; for each parent symbol of the state vertex to be deleted,
; add it to the parent symbol list of the-pair vertex.

                   (do ((rest-sym (par-symbol (state-array cur-vertex-no))
                                  (cdr rest-sym)))
                       ((null rest-sym) nil)
                      (add-par-symbol (car rest-sym) a-prev))
                   (return a-prev))))

            (t (setq pairs  ; else add it to the assoc list
                  (cons (list cur-state (list cur-vertex-no)) pairs))
               (setq return-active (cons an-active return-active))))))


; reduce-all
; inp e.g. ( (1 5 (sh . 4)) (1 5 (re . 3)) (2 10 (sh . 3)))  (structures)
; return all shift active list

(defun reduce-all (active-list lookahead)
   (when (null active-list)
         (return-from reduce-all nil))
   (let ((return-active active-list))

      (do ((a-reduce t) ; exit when this is nil.
           (new-active-list))
          ((null a-reduce) return-active)  ; returns all shift active list
         (multiple-value-setq
            (a-reduce new-active-list)
            (pickup-reduce return-active))
         (when a-reduce
            (setq return-active
               (reduce-one a-reduce new-active-list lookahead))))))


;;; reduce: reduce one active vertex and returns newly-born vertex(es)

(defun reduce-one (a-reduce active-list lookahead)
   (let ((rule (get-rule (cdr (first (active-vertex-action a-reduce))))))
```

```
         return-val)
      (setq return-val
         (backmove-to-symbol
            (active-vertex-vertex-number a-reduce)
            active-list (first rule) (second rule) (third rule)
            nil lookahead nil nil nil))                          ; AL
      return-val))


;;; backmove-to-symbol:  return newly born actives during the session.

(defun backmove-to-symbol (state active-list rule-left rest-rule-right
                           func sons lookahead word-position-list
                           symbol-skip-list symbol-subst-list)          ; AL
   (do ((rest-par (par-symbol (state-array state)) (cdr rest-par))
        (return-actives active-list)
        (a-parent)
        (last-end)
        (last-start)
        (new-skip-list))
       ((null rest-par) return-actives)

      (setq a-parent (car rest-par))

; AL - last-end is the word position in which the parent symbol ends
      (setq last-end (cadr (start-position (symbol-array a-parent))))

; AL - last-start is the word position in which the previous symbol started
      (if (null word-position-list)
          (setq last-start (+ last-end 1))
          (setq last-start (caar word-position-list)))   ; else

      (setq new-skip-list
         (reverse (do ((pos (+ last-end 1) (+ pos 1))
                       (skip-list nil)
                       (skiped-word))
                      ((= pos last-start) skip-list)
                    (setq skiped-word (reverse (assoc pos *input-list*)))
                    (setq skip-list (cons skiped-word skip-list)))))

      (setq new-skip-list (append new-skip-list symbol-skip-list))

      (setq return-actives
         (backmove-to-state a-parent return-actives rule-left rest-rule-right
            func sons lookahead
            (cons (start-position (symbol-array a-parent)) word-position-list)
            (append (skip (symbol-array a-parent)) new-skip-list)
            (append (subst-list (symbol-array a-parent)) symbol-subst-list)))))


;;; backmove-to-state

(defun backmove-to-state (symbol active-list rule-left rest-rule-right func
                          brothers lookahead word-position-list
```

```
                          symbol-skip-list symbol-subst-list)          ; AL
   (cond ((null (cdr rest-rule-right))
          (apply-rule symbol active-list rule-left func
             (cons (node-ptr (symbol-array symbol)) brothers) lookahead
             word-position-list symbol-skip-list symbol-subst-list))
         (t
            (do ((rest-par (par-state (symbol-array symbol)) (cdr rest-par))
                 (return-actives active-list)
                 (a-parent))
                ((null rest-par) return-actives)
               (setq a-parent (car rest-par))
               (setq return-actives
                  (backmove-to-symbol a-parent return-actives rule-left
                     (cdr rest-rule-right) func
                     (cons (node-ptr (symbol-array symbol)) brothers)
                     lookahead word-position-list
                     symbol-skip-list symbol-subst-list))))))


;;; apply-rule

(defun apply-rule (symbol active-list rule-left func sons lookahead
                   word-position-list symbol-skip-list symbol-subst-list) ; AL
   (let (value new-node)
      (cond ((null func) (setq value nil))
            (t (setq value (get-value func sons))
               (when (null value)
                  (return-from apply-rule active-list)))) ; killed
      (setq new-node (push-node rule-left (list sons) value))
      (do ((rest-par (par-state (symbol-array symbol)) (cdr rest-par))
           (a-parent) (new-state-no) (adding-actives nil) (merge-to)
           (new-symbol) (new-active) (active-level) (prev-symbol))
          ((null rest-par) (append active-list adding-actives)) ; return

         (setq a-parent (car rest-par))
         (setq new-state-no
            (get-goto-state (state (state-array a-parent)) rule-left))

         (cond ((setq merge-to (same-destination new-state-no adding-actives))
                  (setq prev-symbol
                     (car (par-symbol (state-array merge-to))))
                  (add-par-state a-parent prev-symbol))
               (t (setq new-symbol
                     (push-symbol rule-left new-node (list a-parent)
                        (list (caar word-position-list)
                              (second (car (last word-position-list))))
                        symbol-skip-list symbol-subst-list))

                  (setq *last-lhs* rule-left)

                  (unless (ambiguity-packing new-symbol active-list
                                             new-state-no lookahead)
                     (setq new-active
                        (make-active-vertex
```

```
                              :vertex-number (push-state new-state-no
                                                         (cadar (last word-position-list))
                                                         (list new-symbol))
                              :level (cadar (last word-position-list))
                              :state new-state-no
                              :action nil))

; AL - add the new active node to the saved-active-list

                        (setq active-level (active-vertex-level new-active))
                        (setf (aref *saved-active-array* active-level)
                           (cons (copy-one new-active)
                                 (aref *saved-active-array* active-level)))

                        (setq adding-actives
                           (append
                              adding-actives
                              (fill-actions (list new-active) lookahead)))))))))


;;; ambiguity-packing
;;;    1. see if the new-symbol is packable to another symbol
;;;       (the same destination, the same origin and the same category)
;;;    2. merge them into the node
;;;   return node to which it merged if packed; return nil otherwise.
;;;   8/5/87 add no-duplicate son check for failure recovery
;;;
;;; AL - logic is refined for handling ambiguities due to skipped words
;;; When a local ambiguity has been detected, skipped word information of the
;;; two symbols is compared :
;;; case 1 : both have same skipped words ---> pack (a real ambiguity !)
;;; case 2 : new symbol has more skipped words ---> new symbol is discarded
;;; case 3 : old symbol has more skipped words ---> old symbol is discarded
;;;          and replaced by the new symbol
;;; case 4 : neither one subsumes the other : they are NOT packed together

(defun ambiguity-packing (new-symbol active-list new-state-no lookahead)
  (let ((saved-active-list nil)
(active-level nil))
     (if (and *no-ambiguity-packing*
              (or (not (eq *skip-word-limit* 1))                          ; AL
                  (not (eq (category (aref *symbol-array* new-symbol))
                           *start-symbol*))))
         (return-from ambiguity-packing nil))

    (setq active-level (second (start-position (symbol-array new-symbol))))
    (setq saved-active-list (aref *saved-active-array* active-level))

    (do ((rest saved-active-list (cdr rest))  ; check for each active
         (an-active)
         (left-state0 (car (par-state (symbol-array new-symbol))))
         ; origin state of the adding symbol. must be only 1 element list.
         (new-node (node-ptr (symbol-array new-symbol)))
         (right-state))
```

```
        ((null rest) nil)
       (setq an-active (car rest))
       (setq right-state
          (same-destination new-state-no (list an-active)))
       (when (and right-state
                  (active-match-p an-active active-list lookahead))
          (do ((rest-par-sym (par-symbol (state-array right-state))
                             (cdr rest-par-sym))
               (a-par-symbol))
              ((null rest-par-sym) nil)
             (setq a-par-symbol (car rest-par-sym))
             (when (equal (category (symbol-array a-par-symbol))
                          (category (symbol-array new-symbol)))

; found a symbol with the same category
; So check whether it has the same origin.
                (do ((rest-par (par-state (symbol-array a-par-symbol))
                               (cdr rest-par))
                     (a-left-state))
                    ((null rest-par) nil)
                   (setq a-left-state (car rest-par))
                   (when (equal left-state0 a-left-state)
                                                ; origin is also the same

;  AL - pack only if symbols have same substitution lists

                      (when (equal (subst-list (symbol-array new-symbol))
                                   (subst-list (symbol-array a-par-symbol)))

;  AL - logic for skipped words in symbols :

;  case 1 : equal skip lists

      (cond ((and (equal (skip (symbol-array new-symbol))
                         (skip (symbol-array a-par-symbol)))
                  (equal (start-position (symbol-array new-symbol))
                         (start-position (symbol-array a-par-symbol))))
             (append-new-ambiguity a-par-symbol new-symbol)
             (delete-par-symbol new-symbol right-state)
             (return-from ambiguity-packing a-par-symbol))

; case 2 : new skip list is bigger

            ((and (is-subset (skip (symbol-array a-par-symbol))
                             (skip (symbol-array new-symbol)))
                  (pos-covers (start-position (symbol-array a-par-symbol))
                              (start-position (symbol-array new-symbol))))
             (return-from ambiguity-packing t))

; case 3 : old skip list is bigger

            ((and (is-subset (skip (symbol-array new-symbol))
                             (skip (symbol-array a-par-symbol)))
                  (pos-covers (start-position (symbol-array new-symbol))
```

```
                                (start-position (symbol-array a-par-symbol))))

; replace old symbol with new-symbol only if new-node hasn't been already
; packed. Otherwise, both will remain at this stage.

              (when (not (listp (node-category (node-array new-node))))
                 (delete-par-symbol a-par-symbol right-state)
                 (add-par-symbol new-symbol right-state)
                 (return-from ambiguity-packing t)))

; case 4 : skip lists do not subsume one another

              (t nil))))))))))))


; same-destination
;     see if the same state number appears in the active list.
;     return the state-vertex-number or nil

(defun same-destination (state-no active-list)
   (do ((rest active-list (cdr rest))
        (an-active))
       ((null rest) nil)
       (setq an-active (car rest))
       (when (= state-no (active-vertex-state an-active))
          (return (active-vertex-vertex-number an-active)))))


;;; shift-all
;;;  ( (1 5 (sh . 4)) (1 5 (sh . 3)) (2 10 (sh . 3))) --> ((3 7 nil)(5 12 nil))

(defun shift-all (shift-active-list word-position)
   (do ((rest shift-active-list (cdr rest))
        (new-active-list nil)
        (right-states nil)         ; assoc ((state-number vertex-number))
        (a-pair)
        (an-active)
        (a-look))
       ((null rest) new-active-list)    ; return
      (setq a-pair (car rest))
      (setq an-active (first a-pair))
      (setq a-look (second a-pair))

; the following is to accept shift-shift conflict
      (do ((shifts-left (active-vertex-action an-active) (cdr shifts-left))
           (right-state-number)
           (same-state))
          ((null shifts-left) nil)
         (setq right-state-number (cdr (car shifts-left)))

         (cond ((setq same-state
                   (second (assoc a-look                                    ; AL
                       (second (assoc right-state-number right-states)))))
                (add-par-state (active-vertex-vertex-number an-active)
```

```
                    (first (par-symbol (state-array same-state)))))

                (t                                                  ; else
                   (multiple-value-bind (new-active pair)
                      (simple-shift (active-vertex-vertex-number an-active)
                                    a-look right-state-number word-position
                                    (first (car shifts-left)))
                      (setq pair                                          ; AL
                         (list (car pair)
                               (cons (second pair)
                                     (second (assoc (car pair) right-states)))))
                      (push pair right-states)
                      (push new-active new-active-list)))))))


; simple-shift    shift a symbol and a state.
; returns a new active vertex and (state-no vertex) assoc list
; add kind to distinguish sh (*shift*) and sh* (*wild-card-shift*)

(defun simple-shift (left-vertex lookahead right-state-number word-position
                     kind)
   (let ((new-symbol nil)
         (right-vertex nil)
         (node-value nil)
         (orig-word nil)
         (cur-level (caar *input-list*)))

      (if (equal kind *wild-card-shift*)
          (setq node-value (list (list 'value lookahead)))
          (setq node-value nil))

      (setq orig-word (assoc word-position *input-list*))             ; AL
      (if (equal (second orig-word) lookahead)
          (setq orig-word nil)
          (setq orig-word
             (list (list word-position
                         (list (second orig-word) lookahead)))))

      (setq new-symbol
         (push-symbol-node lookahead (list left-vertex) nil node-value
   (list word-position word-position)
                           nil orig-word))                            ; AL
      (setq right-vertex
         (push-state right-state-number word-position (list new-symbol)))
      (values (make-active-vertex
                :vertex-number right-vertex
                :level cur-level
                :state right-state-number
                :action nil)  ;to be filled later
              (list right-state-number (list lookahead right-vertex))))) ; AL
```

A.4. Utility Functions

```
; AL - functions for copying an active vertex list
; copy-active-list copies all active vertexes

(defun copy-active-list (active-list)
    (mapcar #'(lambda (x) (copy-one x)) active-list))

; copy-one  does the actual copying of an active-vertex

(defun copy-one (an-active)
   (let (new-active)
        (setq new-active (copy-active-vertex an-active))
        (setf (active-vertex-action new-active) nil)
        new-active))

; AL - skip word functions added
; on-list checks if a skip list is on a list of lists

(defun on-list (skip1 skip-list)
   (cond
      ((null skip-list) nil)
      ((equal skip1 (car skip-list)) skip1)
      (t (on-list skip1 (cdr skip-list)))))

; is-subset checks if list1 is a subset of list2
; needed because subsetp only works on simple lists

(defun is-subset (list1 list2)
   (cond ((null list2) nil)          ;; *AL* correct bug 01/31/94
         ((null list1) t)
         ((on-list (car list1) list2) (is-subset (cdr list1) list2))
         (t nil)))

; pos-covers checks if pos-sym1 covers pos-sym2
; pos-sym1 covers pos-sym2 if it covers at least the same span of words

(defun pos-covers (pos-sym1 pos-sym2)
   (cond ((and (<= (car pos-sym1) (car pos-sym2))
               (>= (second pos-sym1) (second pos-sym2))) t)
         (t nil)))

; is-maximal-parse checks if the current parse is a maximal one

(defun is-maximal-parse (a-sym min-skip)
   (let (sym-skip sym-span full-sym-skip)
      (setq sym-skip (skip (symbol-array a-sym)))
      (setq sym-span (start-position (symbol-array a-sym)))
      (setq full-sym-skip (full-skip sym-skip sym-span))
      (if (= (length full-sym-skip) min-skip)
          (return-from is-maximal-parse t)
          (return-from is-maximal-parse nil))))
```

```
; is-noise determines if the word is a noise word

(defun is-noise (word)
   (equal (char (string word) 0) #\%))


;;;***************************************************************************
;;; Post-parsing processing for display of results
;;;***************************************************************************

; disp-parses displays the final parses

(defun disp-parses (active-list time)
   (let ((value-list nil)
         (score-list nil)
         (symbol-list nil)
         (display-cnt 0)
         (a-sym nil)
         (sym-score nil)
         (sym-span nil)
         (sym-subst nil)
         (sym-skip nil)
         (skip-struct-list nil)
         (sym-sent nil)
         (sym-quality nil)
         (sym-value nil)
         (disp-values nil)
         (big nil)
         (small nil))

      (setq symbol-list (get-symbol-list active-list))
      (setq score-list (score-parses symbol-list))

      (dolist (symbol-pair score-list value-list)

          (when (= display-cnt *display-limit*)
                (setq *parse-value* value-list)
                (return-from disp-parses (car value-list)))

          (setq a-sym (car symbol-pair))
          (setq sym-score (second symbol-pair))
          (setq sym-skip (skip (symbol-array a-sym)))
          (setq sym-subst (subst-list (symbol-array a-sym)))
          (setq sym-span (start-position (symbol-array a-sym)))

          (setq sym-skip (full-skip sym-skip sym-span))
          (setq sym-sent (get-parsed-sent sym-skip sym-subst))
          (setq *skipped-words* sym-skip)
          (setq *subst-words* sym-subst)

          (setq sym-quality (qual-parse sym-score))

          (setq display-cnt (+ display-cnt 1))
```

```
         (format *out* "~2& Parse of input sentence :")
         (format *out* "~&   ~A " sym-sent)
         (format *out* "~2& Parse score is : ~D " sym-score)
         (format *out* "    Parse quality is : ~A " sym-quality)
         (format *out* "~2& Words skipped : ~A " sym-skip)
         (format *out* "~2& Words substituted : ~A " sym-subst)

         (cond (*parser-failed*
                  (format *out*
                      "~2& *** failed but recovered and got the following
                       ***~&"))
               ((> (length sym-value) 1)
                  (format *out* "~2& ~D (~D) ambiguities found and"
                     (length sym-value) (length disp-values)))
               (t
                  (format *out* "~2& ~D (~D) ambiguity found and"
                     (length sym-value) (length disp-values))))

         (multiple-value-setq (big small)
            (floor time internal-time-units-per-second))
         (format *out* " took ~D.~D seconds of real time" big small))

      (return-from disp-parses score-list)))


; full-skip creates the full list of skipped words, including edges

(defun full-skip (skip-list span)
   (let ((temp-skip nil))
      (do ((i 1 (+ i 1))
           (a-word))
          ((= i (car span)) nil)
          (setq a-word (assoc i *input-list*))
          (setq temp-skip
             (append temp-skip (list (list (second a-word) (first a-word))))))

      (setq temp-skip (append temp-skip skip-list))

      (do ((i (+ 1 (second span)) (+ i 1))
           (a-word))
          ((= i (caar *input-list*)) temp-skip)
          (setq a-word (assoc i *input-list*))
          (setq temp-skip
             (append temp-skip
                (list (list (second a-word) (first a-word)))))))))

; get-parsed-sent creates the list of the sentence that was actually parsed

(defun get-parsed-sent (skip-list subst-list)
   (let ((sent nil) found)
      (dolist (a-word *input-list* sent)
         (setq found (assoc (car a-word) subst-list))
         (when (not (null found))
```

```
            (setq a-word (list (car a-word) (second (second found)))))
         (setq found (on-list (reverse a-word) skip-list))
         (when (null found)
            (setq sent (cons (second a-word) sent))))
      (setq *parsed-sent* sent)))


;; get-symbol-list retrieves the list of top "start" symbols from the
;;                 active-list.

(defun get-symbol-list (active-list)
   (let ((symbol-list nil)
         (state-vertex nil)
         (par-sym-list nil))

      (dolist (an-active active-list symbol-list)
         (setq state-vertex (active-vertex-vertex-number an-active))
         (setq par-sym-list (par-symbol (state-array state-vertex)))
         (setq symbol-list (append symbol-list par-sym-list)))))


;; score-parses creates an sorted association list of symbol nodes and their
;;              ranked scores. This list is then used in disp-parses to
;;              display the best parse result(s)

(defun score-parses (symbol-list)
   (let ((cur-sym-pos nil)
         (cur-sym-skip nil)
         (cur-sym-subst nil)
         (full-sym-skip nil)
         (best-count nil)
         (parse-score nil)
         (score-list nil))

; go over all parses and score them, building the assoc list

        (dolist (a-sym symbol-list score-list)
           (setq cur-sym-pos (start-position (symbol-array a-sym)))
           (setq cur-sym-skip (skip (symbol-array a-sym)))
           (setq cur-sym-subst (subst-list (symbol-array a-sym)))
           (setq full-sym-skip (full-skip cur-sym-skip cur-sym-pos))
           (setq best-count (get-best-count a-sym))
           (setq parse-score (+ (length full-sym-skip)
                                (length cur-sym-subst)
                               best-count))
           (setq score-list
             (cons (list a-sym parse-score) score-list)))
        (setq score-list
           (sort score-list #'(lambda (x y)
                                (< (second x) (second y)))))))


;; qual-parse determines the quality of a parse - GOOD/BAD
```

```
(defun qual-parse (score)
   (cond
      ((>= score 8) 'BAD)
      ((and (> score 3)
            (>= (/ score (- (length *input-list*) 1)) '1/2)) 'BAD)
       (t 'GOOD)))


(defun acknowledge ()
"
-------------------------------------------------------------------------------
The Generalized LR Parser/Compiler Version 9.0 is the latest version of the
parsing system, that puts together GLR*, the new robust version of the parser,
with all previous parsing and generation features. The GLR parser has served
as a core component of various projects at the Center for Machine Translation
at Carnegie Mellon University. GLR*, featured in this latest version of the
system is a key component in the JANUS speech-to-speech translation project.
The parser is based on Tomita's Generalized LR Parsing algorithm, augmented by
pseudo and full unification packages. The Generalized LR Parser/Compiler V9.0
is implemented in Common Lisp and no window graphics are used; thus the system
is transportable, in principle, to any machines that support Common Lisp.
Presently, the system runs on Sun SparcStations, HP 9000s running HP-UX 9.x,
DECstations running Mach and IBM RT PC's.

The GLR Parser/Compiler Version 9.0 is available for non-profit and academic
research, subject to the signing of an appropriate licensing agreement.
Those who are interested in obtaining this software system should contact:

    Dr. Alon Lavie
    Center for Machine Translation
    Carnegie-Mellon University
    Pittsburgh, PA 15213, USA
    (412)268-5655
    lavie@cs.cmu.edu

Many members of CMU Center for Machine Translation have made contributions to
the system development. Version 9.0, featuring the GLR* robust version of the
parser was developed by Alon Lavie. The original GLR runtime parser was
implemented by Hiroyuki Musha, Masaru Tomita and Kazuhiro Toyoshima. The
compiler was implemented by Hideto Kagamida and Masaru Tomita. The pseudo
unification package and the full unification package were implemented by
Masaru Tomita and Kevin Knight, respectively. GENKIT and TRANSKIT were
developed by Masaru Tomita and Eric Nyberg. Steve Morrisson, Hideto Tomabechi,
and Hiroaki Saito have also made contributions in maintaining the system.
Some early sample English grammars were developed by Donna Gates, Lori Levin
and Masaru Tomita. A sample Japanese grammar was developed by Teruko Mitamura.
Other members who made indirect contributions in many ways include Kouichi
Takeda, Marion Kee, Sergei Nirenburg, Ralf Brown, and especially Jaime
Carbonell, the director of the center.

Funding for this project is provided by several private institutions and
governmental agencies in the United States and Japan.
-------------------------------------------------------------------------------
")
```

```
(print "Type (ACKNOWLEDGE) for acknowledgements.")
```

Appendix B

Data Sets Used for Performance Evaluations

This appendix contains two of the data sets used in various evaluations presented throughout this thesis. The two sets included here are: (1) the benchmark for runtime performance evaluation data set (see Chapter 3), and (2) the ATIS test set, used in evaluations in Chapters 4, 5 and 6. Two additional test data sets - the JANUS English evaluation test set, and the JANUS/Enthusiast Spanish evaluation test set, are not included here due to their large volume. All four data sets can be obtained electronically on the Internet via anonymous FTP. Instructions on how to do so follow.

B.1. Instructions for Retrieving Data via FTP

The four data sets used for evaluations in this thesis can be obtained electronically on the Internet via anonymous FTP. To do so, follow these instructions:

1. ftp to the host "`ftp.cs.cmu.edu`".
2. login as user-id "anonymous" using your email address as the password.
3. move to the data directory by executing the command: "`cd user/alavie/data`".
4. execute the command: "`get` <*data-set-name*>", where <*data-set-name*> is the name of the desired data set file.
5. execute the command: "`quit`", to exit the ftp program

The names of the retrievable data set files are the following:

- "**benchmark.txt**": the benchmark for runtime performance evaluation data set (which is included in this appendix).
- "**atis-test.txt**": the ATIS test set (included in this appendix).
- "**english-test-trans.txt**": the transcribed version of the JANUS English evaluation test set.
- "**english-test-sp.txt**": the speech recognized version of the JANUS English evaluation test set.
- "**spanish-test-ptt.txt**": the transcribed version of the JANUS/Enthusiast Spanish push-to-talk evaluation test set.

- "**spanish-test-xt.txt**": the transcribed version of the JANUS/Enthusiast Spanish cross-talk evaluation test set.

B.2. Benchmark for Runtime Performance Evaluation

The following set of utterances is the benchmark data set that was used to evaluate the actual time and space performance of GLR* parser in Chapter 3 of the thesis. This data set is a subset of the ESST development set data. The development set data consists of transcribed text of actual recorded dialogs, where the parties attempted to schedule a meeting. The transcribed utterances are broken into sentences. The benchmark set contains 552 of these sentences.

```
(monday and tuesday in fact monday tuesday wednesday and thursday of
 that week nine to twelve i+m out $)
(i have something ten to noon $)
(because i+m booked from nine and four thirty $)
(i think that the conclusion is that week is out $)
(i have something from ten to five $)
(i+m pretty much booked up on tuesday and wednesday $)
(i+m booked up that day $)
(that+s no good $)
(i+m booked up with a seminar for the twenty sixth twenty seventh and
 twenty eighth $)
(my mornings are pretty busy $)
(i+m busy friday through next tuesday $)
(i+m really busy next week $)
(i+m busy until one $)
(as a matter of fact i won+t have any time $)
(let+s see on monday i have something from one to four $)
(friday the eighth of october i seem to only have one hour eleven
 thirty to twelve thirty taken up $)
(friday i+m busy in the afternoon $)
(tuesday and wednesday i+m gonna be out of town $)
(i+m out of town let+s see next wednesday through friday so through
 that monday $)
(+cause i+ll be out of town myself $)
(i+m out of town from the thirtieth to the third $)
(i+ll be out of town from the ninth through the eleventh $)
(i have class from nine to twelve $)
(i got a meeting from three to four thirty $)
(i+m in a seminar $)
(i have a meeting till twelve $)
(i have a meeting from one to two $)
(thursday the twenty ninth i have a meeting from nine to twelve $)
(i got a meeting from one to four $)
(i have a meeting at ten am $)
(i already have a meeting between two and four on monday the eighth $)
```

(i have a meeting from eight am to five pm a seminar $)
(i+ve got a seminar on march the seventeenth $)
(i have class from nine to twelve on the sixteenth $)
(i+ve got a seminar on monday and tuesday $)
(it runs from nine to four thirty $)
(let+s see that means that wednesday+s probably the best for me $)
(tuesday i+m free in the afternoon $)
(and wednesday i+m free all day $)
(but i guess i+m available let+s see late afternoon on friday $)
(and the following tuesday and wednesday i+m pretty free $)
(i+m free in the morning though $)
(but monday i have free $)
(i have friday totally free $)
(between ten and twelve is good $)
(and between one and three is good $)
(that+s good with me $)
(after two looks fine with me $)
(i have friday completely free too $)
(but after twelve o+clock i+m free $)
(that sounds fine with me $)
(next week looks good for me $)
(i+m totally free on tuesday $)
(if you want to meet thursday which is the third $)
(i+m free after ten am $)
(that sounds great $)
(wednesday afternoon sounds good $)
(but after that i+m free $)
(august fourth probably after one would be great for me $)
(between two and four on monday the eighth sounds really good for me $)
(i+ll be available on march the seventeenth $)
(so anytime before nine and anytime after twelve would be fine $)
(that sounds good $)
(i+m free the whole afternoon $)
(i+m free $)
(can you meet in the morning $)
(so how does monday the twenty ninth after three o+clock sound $)
(how does thursday after eleven o+clock sound $)
(how does wednesday february third sound $)
(how+s about monday july twenty six after twelve o+clock sound $)
(do you have a couple hours off during the next couple of weeks $)
(do you have any time after two on the eighth $)
(would you have eleven to one on the twelfth which is friday $)
(would you have anything on tuesday afternoon the sixteenth $)
(do you have an agenda for that day $)
(we have to fit it in there if possible $)
(so what about february the second on tuesday $)
(so let+s jump to tuesday the twenty eighth $)

(but we can meet from eight to ten $)
(if you don+t mind it that early $)
(then let+s plan it for then $)
(on the eighth eight to ten $)
(how +bout next week $)
(how +bout after two $)
(how +bout thursday the twenty ninth $)
(so what about friday $)
(how about wednesday august fourth after one o+clock $)
(should we say between two and four on monday the eighth $)
(let+s try to work something else out $)
(how +bout that $)
(should we say from one pm on on the sixteenth $)
(let+s make it then $)
(hello $)
(hey $)
(take care $)
(thanks $)
(arthur $)
(jackie $)
(jackie $)
(ava $)
(john $)
(cindy $)
(john $)
(okay $)
(sounds good $)
(okay $)
(okay $)
(sounds good $)
(alright $)
(yeah $)
(no $)
(alright $)
(yeah $)
(alright $)
(no $)
(fine $)
(yeah $)
(okay $)
(right $)
(yeah $)
(alright $)
(sounds great to me $)
(yeah $)
(okay $)
(okay $)

(yeah $)
(sure $)
(okay $)
(right $)
(yeah $)
(great $)
(okay $)
(okay $)
(okay $)
(yeah $)
(no $)
(okay $)
(no $)
(sounds good $)
(okay $)
(okay $)
(in fact i+ll tell you what $)
(let+s see $)
(geez $)
(well let+s see $)
(well $)
(well $)
(i+m sorry $)
(i+m sorry $)
(i+m sorry $)
(maybe tuesday $)
(so let+s say two o+clock $)
(and then after that anytime $)
(so we+ll say friday the eighth eight to ten $)
(friday october eighth eight to ten $)
(thursday the eighth at noon $)
(within the next two weeks $)
(one o+clock $)
(after two wednesday the third $)
(or sometime the following week $)
(two o+clock wednesday fourth $)
(but probably in the afternoon early this week or only on thursday
 next week $)
(noon on june three $)
(either tuesday afternoon or wednesday afternoon $)
(two to four on wednesday august fourth $)
(so where do you stand on that $)
(how +bout your calendar $)
(what do you think $)
(well what day were you looking at in general $)
(what+s good for you $)
(when do you have time $)

(so why don+t we meet at noon on thursday the eighth then $)
(i need to meet with you again $)
(so you want to meet about i dunno two o+clock on wednesday the
fourth $)
(could you make a meeting sometime in the next two weeks perhaps
monday afternoon $)
(before we go i think we should schedule a meeting sometime in the
next two weeks for at least two hours $)
(let+s make it a brunch meeting at denny+s $)
(i+d like to go over a couple of things with you in the next couple
of weeks $)
(but do you think we could meet at noon $)
(how +bout if we get together for a couple hours in the next couple
weeks $)
(we really should get together sometime $)
(do you wanna get together in two weeks $)
(+s+at set $)
(i+ll see you on june third at noon $)
(i+ll see you then $)
(i+ll see you then $)
(by monday i assume you mean monday the twenty seventh $)
(so that+s february the third at two $)
(i+m not sure if you meant that you were busy through tuesday $)
(we+re talking about august four $)
(today is friday the fifth $)
(that+s a wednesday $)
(well if you have eleven thirty to twelve thirty filled up there $)
(so let me see $)
(two hours should be fine $)
(or are you gonna be back on tuesday $)
(what do you think $)
(sounds great to me $)
(okay $)
(let me see $)
(twenty actually july twenty sixth and twenty seventh looks good $)
(the twenty sixth afternoon or the twenty seventh before three pm $)
(geez $)
(i+m out of town the thirtieth through the third $)
(i+m in san francisco $)
(or even next wednesday after one o+clock $)
(actually the twenty sixth and the twenty seventh i+ll be at a seminar
all day $)
(twenty ninth is out $)
(and you+re out of town the thirtieth through the third $)
(if that+s the case we could meet possibly wednesday afternoon $)
(after one sounds good to me $)
(the rest of the week doesn+t look too good $)

(how +bout the fourth at two o+clock $)
(maybe from two to four $)
(do you have any ideas just off the top of your head $)
(right now i really don+t have any ideas $)
(then i+ll see you wednesday at two pm $)
(i+ll send you mail regarding the location $)
(thanks $)
(see you then $)
(bye $)
(that sounds good $)
(patty $)
(what do you think about that $)
(thanks $)
(that sounds good $)
(patty $)
(okay $)
(see you then $)
(bye $)
(well marcy $)
(looks like we need to schedule another meeting in the next couple of weeks $)
(so seems like to me the best time would be in the afternoon $)
(what afternoons are you available in the next couple weeks $)
(okay $)
(i look to be available between two and three on the twenty eighth after twelve on the twenty ninth after two on the thirtieth and let+s see between two and three thirty on the first $)
(and that+s really it for me in the afternoons $)
(okay $)
(so you gave me a couple of what seemed to be one and a half hour times $)
(how +bout two hour times $)
(twenty ninth is not good for me in the afternoon $)
(i do have a seminar all day that day $)
(and the afternoon of the thirtieth is also booked from two thirty to five $)
(no $)
(i don+t have anything longer than an hour and a half all day on the first $)
(let+s see is it every day until twelve that you+re busy $)
(because i have a lot of mornings free $)
(actually on the morning of the sixth $)
(well i don+t have anything scheduled at all on the sixth $)
(so we could do something on the morning of the sixth $)
(how would that be for you $)
(because this i+m out of town on the sixth $)
(wait $)

(i really have to rule out the whole week of the fourth $)
(because i don+t have a two hour block at all that whole week $)
(i really don+t have a two hour block between nine and five that whole week except for in the mornings $)
(oops $)
(on the first i have a meeting from ten to eleven $)
(but we could meet $)
(well no $)
(how would that be $)
(no $)
(sorry $)
(first is still bad $)
(here are the times i actually have $)
(before one on the twenty seventh before twelve on the twenty eighth after twelve on the twenty ninth and after two on the thirtieth $)
(those are the only two hour slots i have $)
(if we don+t find something in that time $)
(maybe we can go to the weekend or something $)
(okay $)
(looks like we+re gonna have to go to the weekend $)
(on saturday i already have an appointment from eleven to one pm on the second $)
(but other than that i+m free $)
(how+s your saturday $)
(that actually sounds good $)
(how +bout one o+clock on the second $)
(okay $)
(one to three $)
(it sounds good $)
(alright $)
(great $)
(see you then $)
(okay $)
(marcy $)
(looks like we need to schedule another two hour meeting here $)
(let+s see $)
(next week the week of the twenty second i have two hour blocks available from three to five on the twenty third from twelve to two on the twenty fifth and either before twelve or after one thirty on the twenty sixth $)
(are any of those times good for you $)
(sorry $)
(no $)
(i have something the afternoon of the twenty third $)
(and i+m out of town the twenty fifth through the twenty seventh $)
(how +bout the week of the twenty ninth $)
(wait $)

(i can tell you right off the bat that friday the third is not good $)
(thursday the second is not good $)
(but any morning twenty ninth thirtieth or first are all good $)
(well unfortunately my mornings the twenty ninth thirtieth and first
 are all full $)
(on those three days $)
(let+s see $)
(i would have time on the twenty ninth between twelve and three on
 the thirtieth after twelve $)
(and i don+t have a two hour block free on the first $)
(so either the twenty ninth or thirtieth $)
(how are those for you $)
(no $)
(sorry $)
(none of those are going to work it looks like $)
(it looks like we might have to go to the weekend again $)
(gee $)
(how does before twelve or after two on saturday the fourth look $)
(well saturday the fourth is free all day for me $)
(say ten to twelve on saturday the fourth $)
(yes $)
(ten to twelve on saturday the fourth looks good $)
(thanks $)
(mr. sullivan $)
(yes $)
(so in the next two weeks when can you meet $)
(well let me check my calendar here $)
(we+re on the twenty fourth $)
(well the twenty seventh i have class from nine twelve $)
(so the afternoon is free $)
(and on the twenty eighth also the twenty ninth is totally full $)
(how do those days look for you $)
(well looks like twenty seventh morning would be great for me $)
(twenty eighth morning would also be fine $)
(twenty ninth is pretty busy $)
(afternoon might work $)
(or very early morning $)
(because twenty seventh and twenty eighth mornings are horrible
 for me $)
(and afternoons are fine $)
(and let+s see on the thirtieth $)
(the thirtieth+s pretty horrible too $)
(how +bout the first full week of october $)
(well i+m really sort of sorry $)
(but it looks like i+ll be busy the entire first week of october $)
(how about the friday the first $)
(well i have a meeting from ten am until eleven pm $)

(other than that i+m free $)
(so when are you free $)
(when did you say you were free on thursday $)
(well on thursday i am free from twelve o+clock until two thirty $)
(unless you want to meet after five o+clock that is $)
(nope $)
(after five will certainly not work for me $)
(how +bout saturday the second $)
(well it is a saturday $)
(but after one pm would be fine $)
(how +bout one thirty $)
(would that work for you $)
(one thirty sounds great $)
(okay $)
(then i will see you at one thirty on saturday +till about three
thirty $)
(wonderful $)
(i+ll look forward to it also $)
(you have a good day $)
(mr. sullivan $)
(when in the next two weeks would be good for you $)
(well it looks like next monday and tuesday in the morning would be
fine $)
(or even wednesday in the morning $)
(after that i+ll be out of town for a bit though $)
(goodness $)
(because i have class monday in the morning from nine until twelve $)
(and tuesday in the morning i+ve got a meeting from ten to twelve $)
(so well $)
(but if you+d be able to meet at eight $)
(i could do it or $)
(wednesday i have class again from nine to twelve $)
(so let+s see if we could work something out here $)
(well i+m very sorry $)
(but eight o+clock just won+t work $)
(would you be free at one o+clock on monday or for that matter at
one o+clock on wednesday $)
(unfortunately no i+m not $)
(well i have a meeting at one thirty on tuesday $)
(and i have class at one thirty on wednesday $)
(i am free at three o+clock on tuesday $)
(and i+m free at three thirty on wednesday $)
(perhaps we could get together then or somewhere around then $)
(nope $)
(how +bout the next week $)
(well let+s see monday morning again i have class from nine until
twelve $)

(and on the thirtieth i have a meeting with the other professors
from ten until twelve $)
(so again i don+t like eight o+clock either $)
(but on the thirtieth which is tuesday i could meet either you from
eight until ten $)
(or anytime after twelve o+clock noon would be fine $)
(would either of those work for you $)
(nope unfortunately they wouldn+t $)
(see there+s always wednesday morning $)
(if wednesday morning doesn+t work out then $)
(let me see well wednesday morning again is no good $)
(i do teach class from nine until twelve and from one thirty until
three thirty $)
(would you have time in the evening $)
(perhaps from five o+clock or six o+clock on on wednesday evening $)
(no $)
(i+m very sorry $)
(but my evenings are generally taken up $)
(how +bout saturday the fourth $)
(well if nothing else will work out $)
(then i think i could put it in on saturday at two thirty $)
(would you like to have lunch $)
(a late lunch would be wonderful $)
(okay $)
(see you at two thirty on saturday $)
(great $)
(i+ll see you then $)
(have a good day $)
(hello $)
(dan $)
(it+s me again $)
(and i+m looking at my schedule $)
(okay $)
(the first of april i+m pretty much busy all day except in the
morning before ten $)
(but that would be uncool $)
(the fifth i+m busy all day also $)
(and the eighth i+m free after ten o+clock $)
(so it looks like the eighth is when we can meet $)
(the eighth sounds good $)
(how +bout well i have lunch at eleven thirty $)
(and then we+ll meet from twelve thirty to two thirty $)
(what do you think $)
(okay $)
(that sounds good to me $)
(i+ll meet you at eleven thirty $)
(and then we+ll do our meeting from twelve thirty to two thirty $)

(bye $)
(okay $)
(danny $)
(we need to schedule another one $)
(seems to me like we should meet on the third wednesday $)
(after two o+clock so $)
(and and we+ll be done by five $)
(because everything else is booked $)
(and i+m out of town $)
(i+m really busy this month $)
(okay $)
(the third sounds good $)
(and we can get done just in time $)
(how +bout you $)
(is that good $)
(yeah $)
(it+s excellent $)
(let+s do it $)
(the third sounds good to me $)
(my god $)
(are we going to meet during thanksgiving $)
(well i+m gonna be out of town from the twenty fifth through
 saturday $)
(which leaves me monday morning and tuesday morning $)
(and yeah $)
(monday tuesday wednesday the mornings are all good for me $)
(let+s see monday the twenty ninth the morning+s good for me $)
(god $)
(no $)
(that+s not gonna work for me $)
(let+s see $)
(or actually monday through friday none of my mornings are good at
 all $)
(how are your afternoons $)
(just barely three to five on monday november twenty ninth $)
(how does the first the week of the twenty second look to you $)
(so i don+t know how your noon to three is on monday the twenty
 second $)
(actually i was referring to the week of the twenty second $)
(well wednesday anything before three is good for me $)
(so wednesday between noon and three $)
(can you fit something in $)
(not really $)
(okay $)
(shall we try for evenings $)
(tuesday my meeting starts at two $)
(so we could go noon to two perhaps on tuesday the twenty third of

november $)
(so are your mornings also busy in the first week of december $)
(yes $)
(they are nine to twelve every day $)
(name your time $)
(two thirty to four thirty $)
(you got it $)
(okay $)
(good $)
(thank you $)
(thank you $)
(but i think we should get together again sometime in the next couple
of weeks $)
(okay $)
(let+s see $)
(today+s the twenty fourth $)
(probably next monday would be bad $)
(but but maybe wednesday the twenty ninth $)
(wednesday+s really bad for me $)
(i+ve got a seminar that goes all day $)
(let+s see $)
(would that work for you $)
(no $)
(i+ll be busy most of the afternoon on thursday $)
(maybe not $)
(maybe next friday $)
(anytime after that would be okay with me $)
(okay $)
(do you think we could do it friday between eleven and twelve thirty $)
(what about the following week $)
(do you have some time then $)
(well let+s see i+ll be out of town from monday through wednesday
that+s the fourth through the sixth $)
(and then thursday and friday the seventh and eighth i+ll be busy
all day too $)
(but i+ll be free that weekend the weekend before that the second
and third $)
(okay $)
(well $)
(i hate to work on the weekend $)
(but i can do it $)
(i+m gonna be playing tennis on the second $)
(and that+ll go until about one in the afternoon $)
(so i could meet you possibly starting around two on saturday $)
(or anytime on sunday after about i dunno three in the afternoon
would be alright $)
(okay $)

(either saturday or sunday is fine for me $)
(let+s do it on saturday $)
(i+m free after one $)
(so maybe we can get together at two fifteen or two thirty or so $)
(yeah $)
(two thirty sounds best to me $)
(so saturday the second at two thirty then $)
(okay $)
(we+ll see you then $)
(okay $)
(and we+re gonna have to schedule a meeting sometime in the next
two weeks $)
(so when are you free next week possibly $)
(next week is pretty good for me $)
(i think next week wednesday afternoon would be the best $)
(or thursday after two o+clock $)
(let me see $)
(i+m looking at my schedule $)
(thursday after two is anywhere from two thirty to five on thursday
fine $)
(yes $)
(i have a meeting from ten to twelve $)
(and then another meeting from one to two $)
(are you free after two o+clock on thursday $)
(okay $)
(well $)
(and i have a meeting from two thirty to five $)
(so thursday+s definitely out of the question $)
(as for wednesday it looks like i+m pretty much booked for the whole
entire day from nine am to four thirty pm $)
(how+s the week after look $)
(are you free on perhaps wednesday $)
(no $)
(unfortunately i+m out of town from monday through wednesday $)
(so any of those days won+t work $)
(and the following week thursday and friday i have a seminar from ten
to five $)
(any times are good next week $)
(let+s see on monday and tuesday i have a class from nine to twelve $)
(but i seem to be free anytime from there on afternoon $)
(i+m free in the morning $)
(but not in the afternoon $)
(are you free saturday october second $)
(let+s see $)
(on saturday i just have tennis from eleven to one pm $)
(but i guess we should check out friday possibly $)
(i just have a meeting from ten to eleven am $)

```
(and possibly you free anytime then $)
(no $)
(friday i+m booked all day with meetings and classes $)
(would you mind working on saturday $)
(saturday sounds fine i suppose $)
(i guess right after i get back from tennis $)
(i+m done by one pm $)
(so i guess we could schedule it anytime after then $)
(sounds fine to me $)
(okay $)
(great $)
(let+s meet from two to four $)
(okay $)
(alright $)
(i guess i+ll just see you then $)
(good bye $)
(have a good day $)
```

B.3. ATIS Test Set

The following set of utterances is the test data set for the ATIS domain that was used for evaluations in Chapters 4, 5 and 6 of the thesis. It consists of 119 utterances.

```
(all flights from atlanta to san francisco $)
(four how long does the first one stop $)
(when does the first one stop $)
(when does this flight leave from denver $)
(when does this flight arrive in denver $)
(i need to fly from philadelphia to san francisco on united $)
(i need the flights from philadelphia to san francisco on american $)
(does this flight stop in dallas $)
(can i fly first class on this flight $)
(when does it arrive in dallas $)
(when does it leaves dallas $)
(when are the first class flights available from philadelphia to
 san francisco $)
(when does the american flight stop $)
(what does d f w $)
(i would like to fly from boston to san francisco $)
(i would like a flight from dallas to philadelphia $)
(i would like to fly from denver to boston $)
(i would like to fly from denver to pittsburgh $)
(i would like to go before twelve noon $)
(i would like to and depart before eleven a m $)
(i would like to depart after three p m $)
(i would like the cheapest flight $)
```

```
(would like breakfast flight $)
(after two p m $)
(one way $)
(what type of aircraft is used for flight twelve oakland $)
(which flight is the cheapest $)
(i would like the first flight two mentioned $)
(i would like a ticket on delta flight one zero zero six $)
(book flight fourteen twenty two one $)
(i would like to fly from pittsburgh to atlanta $)
(after three p m $)
(would like a flight on august twenty fifth $)
(please book flight three eighty one $)
(i+d like to fly from atlanta to denver on august twenty nine $)
(i would like to go before eleven a m $)
(what about the flight $)
(book flight eight twenty one $)
(i+d like cheapest philadelphia to boston december twenty fifth $)
(philadelphia to boston please expensive fare $)
(which of these tell me what the least expensive areas from
 philadelphia to boston on december twenty fifth $)
(from philadelphia to boston departing december twenty fifth ways
 the cheapest fare $)
(i+d like to go from philadelphia to boston one way december
 twenty fifth what days cheapest fare $)
(i+d like to go from philadelphia to boston $)
(what meal is the least expensive flight $)
(one way please $)
(i+d like to go from boston to philadelphia $)
(i+d like only snack only the snack $)
(which flights have only the snack $)
(which flight have snack $)
(which flights do not serve breakfast but do serve snack $)
(i would like to go from philadelphia to boston $)
(i+d like to leave in the afternoon $)
(i+d like flights that arrive before five p m $)
(which of these flights are nonstop $)
(which flights arrive between four thirty and five thirty $)
(which flights arrive between the flights that flight ten $)
(i+d like to go from philadelphia to san francisco $)
(which flights from philadelphia to san francisco stop in dallas $)
(does this flight one american $)
(does this flight have first class seats $)
(i+d like to fly first class $)
(what kind of aircraft is it $)
(find the cheapest flight from boston to philadelphia please $)
(how much is the first class ticket on flight eight one three $)
(show flights from the denver at two washington at leaving at
```

eight four seven a m $)
(okay list flights leaving before nine a m $)
(list flights leaving before nine p m $)
(show flights from philadelphia to baltimore leaving after five p m $)
(what kind of aircraft is the eight forty five flight $)
(show flights from philadelphia to san francisco stopping in dallas $)
(how long is a stopover $)
(how long is a stopover on flight four fifty nine $)
(when does this flight arrive in dallas $)
(what does the first class ticket on that flight cost $)
(what is the fare for that flight leaving eighteen and ten days $)
(what does the seating capacity of airplane $)
(how many passengers does that flight hold $)
(what kind of plane is this flight $)
(i+d like the flight from oakland to boston $)
(where is the stop for flight number three fifty four $)
(how much does that flight cost $)
(what are all the flights from oakland to boston $)
(can you show me the prices of all these flights $)
(what the prices of all the flights from oakland to boston $)
(show me the flights from san francisco two atlanta $)
(are there any other flights from san francisco to atlanta $)
(what flight said there from philadelphia to denver $)
(does flight sixteen thirty one serve a meal $)
(what are all flights from denver to pittsburgh four atlanta $)
(what are the flights from atlanta to pittsburgh $)
(can i have a round trip flight from denver to pittsburgh with a stop
in atlanta $)
(what are the prices for all flights from denver to pittsburgh and
from pittsburgh to atlanta and from atlanta to denver $)
(what other prices from the flights from denver two pittsburgh $)
(what the prices from the flights from pittsburgh to atlanta $)
(i+m looking for a flight from boston to atlanta leaving sometime
towards the end of january it should be one way $)
(i+d like to go from boston to atlanta in january one way $)
(are these daily flights $)
(what+s the fare for flight nine seventy five $)
(what kind of plane is flight nine seventy five from boston to
atlanta $)
(i+d like to flight from chicago to san francisco sometime after six
what kind of plane would that be $)
(i+d like to fly from atlanta to san francisco sometime after six
what kind of plane would that day $)
(i+m going from atlanta to san francisco $)
(i+d like to leave after six $)
(what airplanes are those $)
(can you show me the list the flights again $)

```
(i+d like to spend wednesday and new orleans $)
(i+d really like to spend wednesday in oakland $)
(by one i go to oakland on wednesday $)
(do show me the flights back as well $)
(show me flights back $)
(i+d like to come back as late as possible $)
(what+s the latest flight i can return on $)
(what was the last flight you showed me $)
(list the earliest flight leaving leaving boston going to oakland
 on wednesday $)
(what is the latest flight coming back $)
(what+s the latest flight going back $)
(what+s the last return flight $)
(what is the latest return flight $)
```

Bibliography

[1] M-S. Agnäs, H. Alshawi, I. Bretan, D. Carter, K. Ceder, M. Collins, R. Crouch, V. Digalakis, B. Ekholm, B. Gamback, J. Kaja, J. Karlgren, B. Lyberg, P. Price, S. Pulman, M. Rayner, C. Samuelsson, and T. Svensson. Spoken Language Translator: First-Year Report. Technical Report CRC-043, SRI Cambridge, 1994.

[2] A. V. Aho and S.C. Johnson. LR Parsing. *Computing Surveys*, 6(2):99–124, 1974.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[4] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling, Vol. I: Parsing*. Prentice-Hall, Englewood Cliffs, N.J., 1972.

[5] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Computer Science and Information Processing. Addison-Wesley, 1977.

[6] H. Alshawi (ed.). *The Core Language Engine*. MIT Press, Cambridge, MA, 1992.

[7] L. Baum. An Inequality and Associated Maximization Technique in Statistical Estimation for Probabilistic Functions of Markov Processes. *Inequalities*, pages 1–8, 1972.

[8] S. Billot and B. Lang. The Structure of Shared Forests in Ambiguous Parsing. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics (ACL'89)*, Vancouver, BC, Canada, June 1989.

[9] E. Brill, D. Magerman, M. Marcus, and B. Santorini. Deducing Linguistic Structure from the Statistics of Large Corpora. In *Proceedings of DARPA Speech and Natural Language Workshop*, Hidden Valley, Pennsylvania, June 1990.

[10] J. G. Carbonell and P. J. Hayes. Recovery Strategies for Parsing Extragrammatical Language. Technical Report CMU-CS-84-107, Carnegie Mellon University, Pittsburgh, PA, 1984.

[11] J. A. Carroll. *Practical Unification-Based Parsing of Natural Language*. PhD thesis, University of Cambridge, Cambridge, UK, October 1993. Computer Laboratory Technical Report 314.

[12] Y. Chow and S. Roukos. Speech Understanding Using a Unification Grammar. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'89)*, pages 727–730, 1989.

[13] K. Church. A Stochastic Parts Program and Noun Phrase Parser for Unrestricted Text. In *Proceedings of Second Conference on Applied Natural Language Processing (ANLP'88)*, Austin, TX, 1988.

[14] K. Church and W. Gale. Enhanced Good-Turing and Cat-Cal: Two New Methods for Estimating Probabilities of English Bigrams. *Computer Speech and Language*, 4, 1990.

[15] D. Coppersmith and S. Winograd. Matrix Multiplication via Arithmetic Progressions. In *Proceedings of STOC'87*, pages 1–6. ACM press, 1987.

[16] A. Corazza, R. De Mori, R. Gretter, and G. Satta. Stochastic Context-Free Grammars for Island-Driven Probabilistic Parsing. In *Proceedings of Second International Workshop on Parsing Technologies (IWPT'91)*, pages 210–217, Cancun, Mexico, 1991.

[17] F.L. DeRemer. *Practical Translators for LR(k) Languages*. PhD thesis, MIT, Cambridge, MA, 1969.

[18] F.L. DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, 1971.

[19] J. Earley. An Efficient Context-free Parsing Algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[20] J. D. Fodor and L. Frazier. Is the Human Sentence Parsing Mechanism an ATN? *Cognition*, 8:417–459, 1980.

[21] T. Fujisaki, F. Jelinek, J. Cocke, E. Black, and T. Nishino. A Probabilistic Parsing Method for Sentence Disambiguation. *Current Issues in Parsing Technology*, pages 139–152, 1991.

[22] W. A. Gale and K. W. Church. Poor estimates of context are worse than none. In *Proceedings of DARPA Speech and Natural Language Workshop*, Hidden Valley, Pennsylvania, June 1990.

[23] G. Gazdar, Klein E., G. Pullum, and I. Sag. *Generalized Phrase Structured Grammar*. Blackwell, Oxford, UK, 1985.

[24] P. Geutner, B. Suhm, T. Kemp, A. Lavie, L. Mayfield, A. E. McNair, I. Rogina, T. Sloboda, W. Ward, M. Woszczyna, and A. Waibel. Integrating Different Learning Approaches into a Multi-lingual Spoken Translation System. In *IJCAI Workshop on New Approaches to Learning for Natural Language Processing*, Montreal, Canada, August 1995.

[25] E. A. F. Gibson. *A Computational Theory of Human Linguistic Processing: Memory Limitations and Processing Breakdown*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1991. Technical Report CMU-CT-91-125.

[26] O. Glickman. Using Domain Knowledge to Improve End-to-end Performance in a Speech Translation System. Master's thesis, Carnegie Mellon University, Pittsburgh, PA, May 1995. (unpublished).

[27] I. J. Good. The Population Frequencies of Species and the Estimation of Population Parameters. *Biometrica*, 40:237–264, 1953.

[28] G. Hanrieder and G. Gorz. Robust Parsing of Spoken Dialogue Using Contextual Knowledge and Recognition Probabilities. In *Proceedings of ESCA Workshop on Spoken Dialogue Systems*, Denmark, June 1995.

[29] M. P. Harper, L. H. Jamieson, C. D. Mitchell, G. Ying, S. Potisuk, P. N. Srinivasan, R. Chen, C. B. Zoltowski, L. L. McPheters, B. Pellom, and R. A. Helzerman. Integrating Language Models with Speech Recognition. In *Proceedings of AAAI Workshop on Integration of Natural Language and Speech Processing*, pages 139–145, Seattle, WA, August 1994.

[30] A. Hauenstein and H. Weber. An Investigation of Tightly Coupled Time Synchronous Speech Language Interfaces Using a Unification Grammar. In *Proceedings of AAAI Workshop on Integration of Natural Language and Speech Processing*, pages 42–48, Seattle, WA, August 1994.

[31] P. J. Hayes, A. G. Hauptmann, J. G. Carbonell, and M. Tomita. Parsing Spoken Language: a Semantic Caseframe Approach. In *Proceedings of the 11th International Conference on Computational Linguistics (COLING'86)*, pages 587–592, Bonn, Germany, 1986.

[32] P. J. Hayes and G. V. Mouradian. Flexible Parsing. *Computational Linguistics*, 7(4):232–242, 1981.

[33] R. D. Hipp. *Design and Development of Spoken Natural Language Dialog Parsing Systems*. PhD thesis, Duke University, Dept. of Computer Science, Raleigh-Durham, NC, 1992.

[34] J. R. Hobbs and J. Bear. Two Principles of Parse Preference. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING'90)*, volume 3, pages 162–167, Helsinki, Finland, August 1990.

[35] P. S. Jacobs and L. F. Rau. Integrating Top-down and Bottom-up Strategies in a Text Processing System. In *Proceedings of the Second Conference on Applied Natural Language Processing (ANLP'88)*, pages 129–135, Austin, TX, February 1988.

[36] A. N. Jain. *PARSEC: A Connectionist Learning Architecture for Parsing Spoken Language*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, December 1991. Technical Report CMU-CS-91-208.

[37] F. Jelinek, J. D. Lafferty, and R. L. Mercer. Basic Methods of Probabilistic Context Free Grammars. Research Report RC 16374 (#72684), IBM Research Division, T. J. Watson Research Center, 1990.

[38] K. Jensen, G. E. Heidorn, and S. D. Richardson. Parse Fitting and Prose Fixing (Chapter 5). In *Natural Language Processing: the PLNLP Approach*, pages 53–64, Boston, MA, 1993. Kluwer Academic Publishers.

[39] M. Johnson. The Computational Complexity of GLR Parsing. In M. Tomita, editor, *Generalized LR Parsing*, pages 35–42. Kluwer Academic Publishers, 1991.

[40] R. Kaplan and J. Bresnan. Lexical-Functional Grammar: A Formal System for Grammatical Representation. In *The Mental Representation of Grammatical Relations*, pages 173–281. MIT Press, 1982.

[41] F. Karlsson, A. Voutilainen, J. Heikkila, and A. Anttila. *Constraint Grammar - A Language-Independent System for Parsing Unrestricted Text*. Mouton de Gruyter, 1995.

[42] S. M. Katz. Estimation of Probabilities from Sparse Data for the Language Model Component of a Speech Recognizer. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 35(3), 1987.

[43] M. Kay. Algorithm Schemata and Data Structures in Syntactic Processing. Technical Report CSL-80-12, Xerox PARC, 1980. Reprinted in Grosz, Sparck-Jones and Webber (eds.), *Readings in Natural Language Processing*, Morgan Kaufmann, 1986.

[44] J. P. Kimball. Seven Principles of Surface Structure Parsing in Natural Language. *Cognition*, 2:15–47, 1973.

[45] J. R. Kipps. GLR Parsing in Time $O(n^3)$. In M. Tomita, editor, *Generalized LR Parsing*, pages 43–60. Kluwer Academic Publishers, 1991.

[46] K. Kita and W. Ward. Incorporating LR Parsing into SPHINX. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'91)*, pages 269–272, 1991.

[47] D. E. Knuth. On The Translation of Languages from Left to Right. *Information and Control*, 8(6):607–639, 1965.

[48] A. J. Korenjak. A Practical Method for Constructing LR(k) Processors. *Communications of the ACM*, 12(11):613–623, 1969.

[49] R. Kuhn and R. De Mori. A Cache-Based Natural Language Model for Speech Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(6):570–583, 1990.

[50] S. C. Kwasny and N. K. Sondheimer. Relaxation Techniques for Parsing Grammatically Ill-Formed Input in Natural Language Understanding Systems. *American Journal of Computational Linguistics*, 7(2):99–108, 1981.

[51] L. Lambert. *Recognizing Complex Discourse Acts: A Tripartite Plan-Based Model of Dialogue*. PhD thesis, University of Delaware, Newark, DE, September 1993.

[52] L. Lambert and S. Carberry. Modeling Negotiation Subdialogue. In *Proceedings of the 30th Annual Meeting of the Association of Computational Linguistics (ACL'92)*, Newark, DE, 1992.

[53] B. Lang. Parsing Incomplete Sentences. In *Proceedings of 12th International Conference on Computational Linguistics (COLING'88)*, Budapest, Hungary, 1988.

[54] K. Lari and S. J. Young. The Estimation of Stochastic Context-Free Grammars Using the Inside-Outside Algorithm. *Computer Speech and Language*, 4:35–56, 1990.

[55] A. Lavie. An Integrated Heuristic Scheme for Partial Parse Evaluation. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics (ACL-94)*, pages 316–318, Las Cruces, New Mexico, June 1994.

[56] A. Lavie and M. Tomita. Efficient Generalized LR Parsing of Word Lattices. Presented at Bar-Ilan Symposium on Foundations of Artificial Intelligence (BISFAI'93), June 1993.

[57] A. Lavie and M. Tomita. GLR* - An Efficient Noise Skipping Parsing Algorithm for Context Free Grammars. In *Proceedings of the third International Workshop on Parsing Technologies (IWPT-93)*, pages 123–134, Tilburg, The Netherlands, August 1993.

[58] R. Leech, G.and Garside. Running a Grammar Factory: The Production of Syntactically Analysed Corpora or "Treebanks". In *English Computer Corpora: Selected Papaers and Research Guide (Edited by S. Johansson and A. Stenstom)*, pages 15–32. Mouton de Gruyter, 1991.

[59] J. F. Lehman. *Adaptive Parsing: Self-extending Natural Language Interfaces*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, August 1989. Technical Report CMU-CS-89-191.

[60] L. Levin, O. Glickman, Y. Qu, D. Gates, A. Lavie, C. P. Rosé, C. Van Ess-Dykema, and A. Waibel. Using Context in Machine Translation of Spoken Language. In *Proceedings of Theoretical and Methodological Issues in Machine Translation (TMI-95)*, Leuven, Belgium, July 1995.

[61] D. M. Magerman and M. P. Marcus. Pearl: a Probabilistic Chart Parser. In *Proceedings of Second International Workshop on Parsing Technologies (IWPT'91)*, pages 193–199, Cancun, Mexico, 1991.

[62] D. M. Magerman and C. Weir. Efficiency, Robustness and Accuracy in Picky Chart Parsing. In *Proceedings of the 30th Annual Meeting of the Association of Computational Linguistics (ACL'92)*, pages 40–47, Newark, DE, 1992.

[63] L. Mayfield, M. Gavaldà, W. Ward, and A. Waibel. Concept-based Speech Translation. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'95)*, Detroit, MI, 1995.

[64] M. C. McCord. Heuristics for Broad-Coverage Natural Language Parsing. In *Proceedings of the ARPA Workshop on Human Language Technology*, Princeton, NJ, March 1993. Morgan Kaufmann.

[65] D. D. McDonald. Robust Partial Parsing through Incremental Multi-level Processing: Rationales and Biases. In P. S. Jacobs, editor, *Text-based Intelligent Systems: Current Research in Text Analysis, Information Extraction and Retrieval*, pages 61–65, September 1990. Selected Papers from AAAI Spring Symposium on Text-based Intelligent System, March 1990, Stanford University, Stanford, CA. Published as Technical Report 90CRD198, GE Research and Development Center, Schenectady, NY.

[66] D. D. McDonald. An Efficient Chart-based Algorithm for Partial Parsing of Unrestricted Texts. In *Proceedings of the Third Conference on Applied Natural Language Processing (ANLP'92)*, pages 193–200, Trento, Italy, April 1992.

[67] S. W. McRoy. The Influence of Time and Memory Constraints on the Resolution of Structural Ambiguity. Technical Report CSRI-209, Computer Systems Research Institute, University of Toronto, September 1988.

[68] C. S. Mellish. Some Chart Based Techniques for Parsing Ill-formed Input. In *Proceedings of the 27th Annual Meeting of the Association of Computational Linguistics (ACL'89)*, pages 102–109, 1989.

[69] W. Menzel. Robust Processing of Natural Language. In *Proceedings of 19th German Conference on Artificial Intelligence (KI'95)*, Bielefeld, Germany, 1995. (To appear. Advance copy obtained via the Computational Linguistics Electronic Archive).

[70] R. M. Moore, F. Pereira, and H. Murveit. Integrating Speech and Natural Language Processing. In *Proceedings of the Speech and Natural Language Workshop*, pages 243–247, Philadelphia, Pennsylvania, 1989.

[71] H. Murveit and R. Moore. Integrating Natural Language Constraints into HMM-based Speech Recognition. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'90)*, pages 573–576, 1990.

[72] S. Nakagawa. Speaker Independent Continuous Speech Recognition by Phoneme-based Word Spotting and Time Synchronous Context-free Parsing. *Computer Speech and Language*, 3(3):277–299, July 1989.

[73] T. Nasukawa. Robust Parsing Based on Discourse Information: Completing the Partial Parses of Ill-formed Sentences on the Basis of Discourse Information. In *Proceedings of the 33th Annual Meeting of the Association of Computational Linguistics (ACL'95)*, Boston, MA, June 1995.

[74] H. Ney. Dynamic Programming Speech Recognition Using a Context-free Grammar. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'87)*, pages 69–72, 1987.

[75] S. Ng and M. Tomita. Probabilistic LR Parsing for General Context-Free Grammars. In *Proceedings of Second International Workshop on Parsing Technologies (IWPT'91)*, pages 154–163, Cancun, Mexico, 1991.

[76] R. Nozohoor-Farshi. GLR Parsing for ϵ-Grammars. In M. Tomita, editor, *Generalized LR Parsing*, pages 61–76. Kluwer Academic Publishers, 1991.

[77] M. Okada. A Unification Grammar Directed One Pass Search Algorithm for Parsing Spoken Language. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'91)*, pages 721–724, April 1991.

[78] Giachin E. P. and C. Rullent. Robust Parsing of Severely Corrupted Spoken Utterances. In *Proceedings of 12th International Conference on Computational Linguistics (COLING'88)*, pages 196–201, Budapest, Hungary, 1988.

[79] C. Pollard and I. Sag. *Information Based Syntax and Semantics: Vol. 1 - Fundamentals*. University of Chicago Press, Chicago, IL, 1987.

[80] L. R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. In *Readings in Speech Recognition (Edited by: A. Waibel and K-F. Lee)*, pages 267–296. Morgan Kaufmann, 1990.

[81] C. P. Rosé, B. Di Eugenio, L. S. Levin, and C. Van Ess-Dykema. Discourse Processing of Dialogues with Multiple Threads. In *Proceedings of the 33th Annual Meeting of the Association of Computational Linguistics (ACL'95)*, Boston, MA, 1995.

[82] H. Saito. Bi-directional LR Parsing from an Anchor Word for Speech Recognition. In *Proceedings of 13th International Conference on Computational Linguistics (COLING'90)*, Helsinki, Finland, 1990.

[83] H. Saito and M. Tomita. Parsing Noisy Sentences. In *Proceedings of 12th International Conference on Computational Linguistics (COLING'88)*, Budapest, Hungary, 1988.

[84] Y. Schabes and R. C. Waters. Stochastic Lexicalized Context-Free Grammar. In *Proceedings of the 3rd International Workshop on Parsing Technologies (IWPT'93)*, pages 257–266, Tilburg, The Netherlands, 1993.

[85] S. Seneff. A Relaxation Method for Understanding Spontaneous Speech Utterances. In *Proceedings of DARPA Speech and Natural Language Workshop*, pages 299–304, February 1992.

[86] S. Seneff. TINA: A Natural Language System for Spoken Language Applications. *Computational Linguistics*, 18(1):61–86, 1992.

[87] P. Shann. Experiments with GLR and Chart Parsing. In M. Tomita, editor, *Generalized LR Parsing*, pages 17–34. Kluwer Academic Publishers, 1991.

[88] S. Staab. GLR Parsing of Word Lattices Using a Beam Search Method. To be presented at Eurospeech'95, Madrid, Spain. Advance copy obtained via the Computational Linguistics Electronic Archive, 1995.

[89] D. Stallard and R. Bobrow. Fragment Processing in the DELPHI System. In *Proceedings of DARPA Speech and Natural Language Workshop*, pages 305–310, February 1992.

[90] T. Strzalkowski. TTP: A Fast and Robust Parser for Natural Language. In *Proceedings of International Conference on Computational Linguistics (COLING'92)*, pages 198–204, Nantes, France, August 1992.

[91] B. Suhm, P. Geutner, T. Kemp, A. Lavie, L. Mayfield, A. E. McNair, I. Rogina, T. Sloboda, W. Ward, M. Woszczyna, and A. Waibel. JANUS: Towards Multi-lingual Spoken Language Translation. In *ARPA Workshop on Spoken Language Technology*, 1995.

[92] B. Suhm, L. S. Levin, N. Coccaro, J. G. Carbonell, K. Horiguchi, R. Isotani, A. Lavie, L. Mayfield, C. P. Rosé, C. Van Ess-Dykema, and A. Waibel. Speech-Language Integration in a Multi-lingual Speech Translation System. In *Proceedings of AAAI Workshop on Integration of Natural Language and Speech Processing*, pages 92–98, Seattle, WA, August 1994.

[93] M. Tomita. An Efficient Word Lattice Parsing Algorithm for Continuous Speech Recognition. In *Proceedings of IEEE-IECEJ-ASJ International Conference on Acoustics, Speech and Signal Processing (ICASSP'86)*, pages 1569–1572, Tokyo, Japan, April 1986.

[94] M. Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, Hingham, MA, 1986.

[95] M. Tomita. An Efficient Augmented Context-free Parsing Algorithm. *Computational Linguistics*, 13(1-2):31–46, 1987.

[96] M. Tomita. The Generalized LR Parser/Compiler - Version 8.4. In *Proceedings of International Conference on Computational Linguistics (COLING'90)*, pages 59–63, Helsinki, Finland, 1990.

[97] M. Tomita, T. Mitamura, H. Musha, and M. Kee. The Generalized LR Parser/Compiler - Version 8.1: User's Guide. Technical Report CMU-CMT-88-MEMO, Carnegie Mellon University, Pittsburgh, PA, April 1988.

[98] M. Tomita and E. H. Nyberg 3rd. Generation Kit and Transformation Kit, Version 3.2: User's Manual. Technical Report CMU-CMT-88-MEMO, Carnegie Mellon University, Pittsburgh, PA, October 1988.

[99] L. Valiant. General Context Free Recognition in Less than Cubic Time. *Journal of Computer and Systems Sciences*, 10(2):308–315, 1975.

[100] A. Viterbi. Error Bounds for Convolutional Codes and an Asympyotically Optimal Decoding Algorithm. *IEEE Transactions on Information Theory*, IT-13:260–269, April 1967.

[101] A. Waibel, A. N. Jain, A. McNair, J. Tebelskis, L. Osterholtz, H. Saito, O. Schmidbauer, T. Sloboda, and M. Woszczyna. JANUS: Speech-to-Speech Translation Using Connectionist and Non-Connectionist Techniques. *Advances in Neural Information Processing Systems*, 4, 1991.

[102] K. Ward and D. G. Novick. On the Need for a Theory of Integration of Knowledge Sources for Spoken Language Understanding. In *Proceedings of AAAI Workshop on Integration of Natural Language and Speech Processing*, pages 23–30, Seattle, WA, August 1994.

[103] W. Ward. Understanding Spontaneous Speech: The Phoenix System. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'91)*, pages 365–367, April 1991.

[104] W. Ward, S. Issar, X. Huang, H. Hon, M. Hwang, S. Young, M. Matessa, F. Liu, and R. Stern. Speech Understanding in Open Tasks. In *Proceedings of DARPA Speech and Natural Language Workshop*, pages 78–83, February 1992.

[105] R. M. Weischedel and J. Black. Responding Intelligently to Unparsable Inputs. *American Journal of Computational Linguistics*, 6(2):97–109, 1980.

[106] M. Woszczyna, N. Aoki-Waibel, F. D. Buo, N. Coccaro, T. Horiguchi, K. amd Kemp, A. Lavie, A. McNair, T. Polzin, I. Rogina, C. P. Rosé, T. Schultz, B. Suhm, M. Tomita, and A. Waibel. JANUS-93: Towards Spontaneous Speech Translation. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'94)*, 1994.

[107] M. Woszczyna, N. Coccaro, A. Eisele, A. Lavie, A. McNair, T. Polzin, I. Rogina, C. P. Rosé, T. Sloboda, M. Tomita, J. Tsutsumi, N. Aoki-Waibel, W. Ward, and A. Waibel. Recent Advances in JANUS: a Speech Translation System. In *Proceedings of Eurospeech*, 1993.

[108] J. Wright. LR Parsing of Probabilistic Grammars with Input Uncertainty for Speech Recognition. *Computer Speech and Language*, 4:297–323, 1990.

[109] J. Wright, A. Wrigley, and R. Sharman. Adaptive Probabilistic Generalized LR Parsing. In *Proceedings of Second International Workshop on Parsing Technologies (IWPT'91)*, pages 100–109, Cancun, Mexico, 1991.

[110] J. H. Wright and E. N. Wrigley. Probabilistic LR Parsing for Speech Recognition. In *Proceedings of First International Workshop on Parsing Technologies (IWPT'89)*, pages 105–114, Pittsburgh, PA, 1989.

[111] S. R. Young, A. G. Hauptmann, W. H. Ward, Smith E. T., and P. Werner. High Level Knowledge Sources in Usable Speech Recognition Systems. *Communications of the ACM*, 32(2):183–194, February 1989. Also appears in Waibel and Lee (eds.), Readings in Speech Recognition, Morgan Kaufman, 1990.

[112] D. H. Younger. Recognition and Parsing of Context Free Languages in Time n^3. *Information and Control*, 10:198–208, 1967.

[113] V. Zue, J. Glass, D. Goodine, H. Leung, M. Phillips, J. Polifroni, and S. Seneff. Integration od Speech Recognition and Natural Language Processing in the MIT VOYAGER System. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'91)*, pages 713–716, April 1991.