GLR Parsing with Multiple Grammars for Natural Language Queries

LUK Po Chui
陸寶翠

A Thesis
Submitted in Partial Fulfilment of the Requirements for the Degree of
Master of Philosophy
in
Systems Engineering and Engineering Management

August 2000


香港中文大學
圖書館系統
28 JUN 2001
UNIVERSITY
LIBRARY SYSTEM
THE CHINESE UNIVERSITY OF HONG KONG

Abstract

Parsing natural language (or spoken language) is a challenging task. Natural language is inherently ambiguous – a sentence may have more than one representations. It may also vary significantly in style from speaker to speaker. Consequently, a natural language parser often needs to handle a large number of grammar rules.

This thesis presents an approach for parsing natural language queries, which integrates multiple sub-parsers and sub-grammars; in contrast with the traditional single grammar and parser approach. In using LR(k) processors for natural language processing, we are faced with exponential growth of parsing table sizes with increasing grammar sizes. We propose to partition a grammar into multiple sub-grammars, each having its own parsing table and parser. Grammar partitioning should help reduce the parsing table sizes when compared using a single grammar. We use the GLR parser with an LR(1) parsing table in our framework, because GLR parsers can handle ambiguity. A parser composition technique then combines the parsers' outputs to produce an overall parse which is same as the output parse of single parser. Two different methods were implemented for parser composition – parser composition by cascading and parser composition with predictive pruning.

Our experiments are conducted with natural language queries in the ATIS domain. The unpartitioned English ATIS grammar has 72,869 states in its parsing table, while the partitioned English grammar has 3,350 states in total. This result shows grammar partitioning greatly economizes on the parsing table sizes. Language understanding performances are also examined. Parser composition obtains a higher understanding performance when compare single GLR approach. A contributing factor is partial parses for ungrammatical sentences.

摘要

語法分析(口述語言分析)是一項具挑戰性的工作，自然語言本身是含糊不清的，同一句句子可以有多過一種的表達手法。此外，不同的講者有著不同的說話方式，故此，自然語言分析器經常要處理大量的文法規則。

這篇論文提出一種分析問句的方法，這方法組合了多個語言分析器及文法規則，與傳統的單一語言分析器及文法規則作出對比。在使用LR(k)處理器分析自然語言時，我們需要面對處理大量文法規則所引致的指數增長的分析狀態表。我們提出將一組文法規則分折成多組文法規，每組文法規都各自有自己的分析狀態表，這種方法可以減少分析狀態表的大小。我們使用GLR分析器作爲藍本，因爲它可以處理語意含糊。接著我們使用組合語言分析器技巧來結合所有分析器，而得出一棵同單一分析器所得出來一樣的分析樹。我們實行了兩種不同的分析器組合法，包括級聯式(cascading)及附有預示性修剪的(predictive pruning)組合法。

我們測試了我們的語言分析器在ATIS領域中的問句，發現了未被分割的英語ATIS文法規則具有一個有72,869個狀態的分析狀態表，而被分割的英語ATIS文法規則卻只共有3,350個狀態。由此可見，文法規則分割有效地減少在分析狀態表的狀態。與此同時，我們也測試了語言理解

能力的表現，發現了分析器組合法比起單一語言分析器在處理語法錯誤的句子取得了較好的語言理解能力。

Acknowledgments

I would first like to express my sincere gratitude to my supervisor, Professor Helen Meng, for her support, guidance and encouragement in my thesis work. I would like to thank her for spending a lot of time to work with me on my thesis. She also shared her life experience to me and made me feel fruitful in these two years.

I am also grateful to the members of my thesis committee for giving me advise and reading of my thesis: Dr. Fuliang Weng, Professor Lin-Shan Lee, Professor Kai-Pui Lam, Professor Kam-Fai Wong and Professor Christopher Yang. Special thanks to Dr. Fuliang Weng for arising my interests in parsing, and providing very valuable advice and suggestions on my research.

I would like to thank all my classmates in SEEM. They gave me support and help when I have problems, ease my difficult times and I never feel alone when conducting my research. Many thanks to: Ah Kin, Ah Su, Carmen, Kun-Chung, Timmy, Tony and Wai-Ip. I also want to thank all my colleagues and fellow classmates in the Human-Computer Communications Laboratory for providing a joyful working environment to me: Ah Fan, Brenda, Ida, Connie, Sally, Yuk Chi and Tiffany. To my classmates and officemates Ada Ng, Ah Lai, Jason, Chiu Chun and Jessica, who were considerate and shared

much fun times with me in the office throughout this two years. I also thank all my best friends for sharing with me during my ups and downs.

Special thanks to my family, who always gave me support, take care of me and never blame me for spending less time with them.

Contents

1 Introduction **1**

1.1 Efficiency and Memory 2

1.2 Ambiguity 3

1.3 Robustness 4

1.4 Thesis Organization 5

2 Background **7**

2.1 Introduction 7

2.2 Context-Free Grammars 8

2.3 The LR Parsing Algorithm 9

2.4 The Generalized LR Parsing Algorithm 12

2.4.1 Graph-Structured Stack 12

2.4.2 Packed Shared Parse Forest 14

2.5 Time and Space Complexity 16

2.6 Related Work on Parsing 17

2.6.1 *GLR** 17

2.6.2 TINA 18

2.6.3 PHOENIX 19

2.7 Chapter Summary 21

3 Grammar Partitioning **22**

3.1 Introduction 22

3.2 Motivation 22

3.3 Previous Work on Grammar Partitioning 24

3.4 Our Grammar Partitioning Approach 26
3.4.1 Definitions and Concepts 26
3.4.2 Guidelines for Grammar Partitioning 29
3.5 An Example 30
3.6 Chapter Summary 34

4 Parser Composition **35**
4.1 Introduction 35
4.2 GLR Lattice Parsing 36
4.2.1 Lattice with Multiple Granularity 36
4.2.2 Modifications to the GLR Parsing Algorithm 37
4.3 Parser Composition Algorithms 45
4.3.1 Parser Composition by Cascading 46
4.3.2 Parser Composition with Predictive Pruning 48
4.3.3 Comparison of Parser Composition by Cascading and Parser Composition with Predictive Pruning 54
4.4 Chapter Summary 54

5 Experimental Results and Analysis **56**
5.1 Introduction 56
5.2 Experimental Corpus 57
5.3 ATIS Grammar Development 60
5.4 Grammar Partitioning and Parser Composition on ATIS Domain 62
5.4.1 ATIS Grammar Partitioning 62
5.4.2 Parser Composition on ATIS 63
5.5 Ambiguity Handling 66
5.6 Semantic Interpretation 69
5.6.1 Best Path Selection 69
5.6.2 Semantic Frame Generation 71
5.6.3 Post-Processing 72
5.7 Experiments 73
5.7.1 Grammar Coverage 73
5.7.2 Size of Parsing Table 74

5.7.3 Computational Costs 76
5.7.4 Accuracy Measures in Natural Language Understanding 81
5.7.5 Summary of Results 90
5.8 Chapter Summary 91

6 Conclusions **92**
6.1 Thesis Summary 92
6.2 Thesis Contributions 93
6.3 Future Work 94
6.3.1 Statistical Approach on Grammar Partitioning 94
6.3.2 Probabilistic modeling for Best Parse Selection 95
6.3.3 Robust Parsing Strategies 96

Bibliography **97**

A ATIS-3 Grammar **101**
A.1 English ATIS-3 Grammar Rules 101
A.2 Chinese ATIS-3 Grammar Rules 104

List of Figures

2.1 A model of an LR parser. 11
2.2 The initial stage of a GSS. 13
2.3 The GSS after rule reduction. 15
2.4 The GSS after state shifted. 15
2.5 The GSS after local ambiguity packing. 15
2.6 The packed forest of parse vertex J after local ambiguity packing. 16

3.1 The calling graph of the sub-grammars G_0, G_1 and G_2. 33

4.1 An example of a lattice. 37
4.2 The initial stage of a GSS. 38
4.3 The GSS after rule reductions. 39
4.4 The GSS after shift i. 40
4.5 The input LMG for example in Modification 1 40
4.6 The initial stage of the LMG. 41
4.7 Beginning steps of parser operations in the GSS. 43
4.8 The trace of parser in GSS when it reads $vtNP$ with index 7. 44
4.9 The final stage of the GSS. 44
4.10 The resultant parse forest from sub-parser of G_0. 45
4.11 The control of parser composition by cascading. 47
4.12 Changes in LMG due to sub-parsers in level 0 through parser composition by cascading. 49
4.13 Changes in LMG by parser composition with predictive pruning. 53

5.1 Examples of English sentences in the ATIS-3 training corpus, together with their Cantonese translations. 58

5.2 An example of a SQL query for data base access, together with its English query. 59
5.3 The calling graph of the sub-grammars in Table 5.4. 65
5.4 The resultant LMG by using parser composition by cascading. 65
5.5 The resultant LMG by using parser composition with predictive pruning. 66
5.6 Example of a multiple parses sentence. 67
5.7 Another example of multiple parses sentence. 67
5.8 A subset of nodes in a LMG. 69
5.9 Algorithm for shortest path problem [10]. 71
5.10 The parse tree is attached to a virtual terminal VTDEPARTURE. 72
5.11 Example of a semantic frame. 72
5.12 The semantic frame generated for the query (sentence 1) "i would like to book a round trip flight from kansas city to chicago." This frame fully matches the reference semantic frame in Figure 5.13, and is counted as a "full match" in Table 5.11. 85
5.13 The reference frame for the query (sentence 1) "i would like to book a round trip flight from kansas city to chicago," generated from SQL. 85
5.14 The semantic frame generated for the query (sentence 2) "find american flight from newark to nashville around six thirty p m." This frame fully matches the reference semantic frame in Figure 5.15, and is counted as a "partial match" in Table 5.11. 85
5.15 The reference frame for the query (sentence 2) "find american flight from newark to nashville around six thirty p m." generated from SQL. 86

List of Tables

3.1 LR(1) parsing table for unpartitioned grammar G_S. 31
3.2 LR(1) parsing table for partitioned grammar G_0. 32
3.3 LR(1) parsing table for partitioned grammar G_1. 33
3.4 LR(1) parsing table for partitioned grammar G_2. 33

4.1 The left corner node sets of virtual terminals. 51

5.1 Average length of the ATIS-3 queries for the English and Chinese corpora. 58
5.2 A subset of English ATIS grammar. 61
5.3 Grammar statistics based on the original unpartitioned grammar and partitioned grammar. 62
5.4 A subset of partitioned English ATIS grammar. 64
5.5 Grammar coverage for the English ATIS-3 corpora. 75
5.6 Grammar coverage for the Chinese ATIS-3 corpora. 75
5.7 Computational costs for the English ATIS-3 corpora. Italicized percentages in parentheses are savings of PP relative to CAS. 77
5.8 Computational costs for the Chinese ATIS-3 corpora. Italicized percentages in parentheses are savings of PP relative to CAS. 78
5.9 Computational costs for the subsets of sentences in test set of English ATIS-3 corpora with full parses. Italicized percentages in parentheses are savings of PP relative to CAS. 80

5.10 Computational costs for the subsets of sentences in test set of Chinese ATIS-3 corpora with full parses. Italicized percentages in parentheses are savings of PP relative to CAS. 81
5.11 Performance in language understanding of the English ATIS-3 corpora. 84
5.12 Performance in language understanding of the Chinese ATIS-3 corpora. 84
5.13 Language understanding performance for the subsets of sentences in test set of English ATIS-3 corpora with full parses. . 88
5.14 Language understanding performance for the subsets of sentences in test set of Chinese ATIS-3 corpora with full parses. 88
5.15 Language understanding performance for the subsets of sentences in test set of English ATIS-3 corpora with partial parses. 89
5.16 Language understanding performance for the subsets of sentences in test set of Chinese ATIS-3 corpora with partial parses. 89

Chapter 1

Introduction

Natural language is a primary medium for human-human communication, hence natural language conveys meaning and reflects human thinking. Natural language processing is a desirable means of human-computer interaction, since it provides a natural, intelligible and effective way for human to interact with computer, and requires no specialized learning or training. As a result, natural language processing has become a key research area recently. Parsing is an important technology constituting component of natural language systems. Many practical systems involve parsing technology, such as machine translation, speech recognition, language understanding, etc.

Traditionally, a parser is a component of a compiler. Parsing is the process of constructing a parse tree or hierarchical structure from the terminals or words in a sentence based on a given grammar. The methods commonly used in parsers are top-down or bottom-up. Top-down parsers build parse trees from the top (root) to the bottom (leaves), while bottom-up parsers build parse trees form the leaves and work up to the root. The input to the parser

is scanned from left to right.

However, parsing natural language is a more challenging task than parsing programming languages. First of all, natural language are inherently *ambiguous*, whereas programming language are not. Moreover, natural language (or spoken language) may vary significantly in style from speaker to speaker. Consequently, the natural language parser need to handle a large grammar, even for restricted to a specific domains. However, it is difficult to write a grammar that can fully cover most of natural language, so *robustness* is another key issue. In addition, parsing natural language for a human-computer interface requires a real-time response to the user, so *efficiency* in parsing is a major concern as well. *Memory* is closely related to *efficiency* in parsing technology. Large memory needed parser is not only expensive for data storage, and also time consuming to access data.

Efficiency, *memory*, *ambiguity* and *robustness* are the central issues in natural language parsing. Different parsers have exhibited different characteristics with respect to these issues. Some researchers focus on [8][18], while others look for alternative ways to tackle these difficult parsing issues. Modular parsing architectures also receive much attention [1][5][6]. We also focus on these several aspects in developing our parsing framework. the performance of particular parsing algorithms

1.1 Efficiency and Memory

We choose to begin with an implementation of LR parsing, because it is easy to implement and computationally efficient. Parsing a sentence of length N

can be accomplished in $O(N)$ time and space. We will review the principle of LR parsing algorithm in Chapter 2.

In a practical natural language system, a parser need to handle many variations in a set of grammar rules. The grammar grows due to variability for different users (or speakers if we are parsing spoken language from speech recognition). For instance, there are many different ways to express a date, such as march first, first of march, etc. Users' queries often start with "i need", "may i", "i would like to know", etc. However, the computation required for constructing LR(k) parsing table increases rapidly with the size and complexity of the grammar and with the value of k. Earley [7] shows a class of grammars for which the number of states grow exponentially with grammar size. In order to handle a large set of grammar, we propose a *grammar partitioning* approach in this thesis. We partition the entire grammar into a number of sub-grammars. This reduces total number of states (rows) of parsing table and hence computation time for generating the parsing table. A smaller size of parsing table can save space for storage. The parsing architecture is modular, since all parsing tables are constructed separately. Hence if any single sub-grammar needs to be modified, we only need to regenerate the corresponding parsing table. This eases the development of natural language parsers, because grammar can be reused and incorporated easier.

1.2 Ambiguity

A sentence is ambiguous if it has multiple parses. Ambiguity is nonexistent in programming languages, a sentence must always have only one parse.

However, ambiguity is prevalent in natural language. The problem is worse if only syntactic grammar rules are used. Consider the famous example "i saw a man with telescope". The prepositional phrase "with a telescope" can be attached to the sentence phrase "i saw a man" or the noun phrase "a man". Ideally, the parser should only produce one parse out of an ambiguous sentence. However, some sentences are ambiguous, that even a human cannot disambiguate them. For example, "new york" can represent a city name or a state name. Therefore, it is not acceptable for a practical natural language parser to produce only one arbitrary parse. The GLR parsing algorithm is able to handle ambiguous grammars. Hence, we adopt the GLR parsing algorithm is instead of LR parsing algorithm in our work. In Chapter 2, we will to describe the GLR parsing algorithm in detail.

1.3 Robustness

Due to writing or speaking errors, natural language parser usually need to deal with extra-grammatical sentences. A robust parser should parse the core part of the sentence, and extract the coherent meaning from these imperfect sentences.

GLR parsing algorithm is designed to analyze grammatical input sentence. It can detect ungrammatical input as soon as possible, but fail to parse any input which is found to be ungrammatical. Sentences that are grammatical in the large sense may fail to be completely covered by the grammar. We try to release the strict syntactical condition of GLR parsing.

After entire grammar is partitioned into small parts of sub-grammar,

parser composition algorithm is needed to integrate sub-parsers for parsing. *Parser composition by cascading* – one of parser composition algorithm, is able to deal with extra-grammatical queries. It is because it allows all sub-parsers to start and end at any position of the input sentence. However, it has a problem of over generation of semantic and syntactic structure during parsing. *Parser composition with predictive pruning* – another parser composition algorithm, gives an improvement in parsing speed at the expense of less robustness.

1.4 Thesis Organization

This thesis explores the issues introduced above and evaluates the parsing performance based on these issues. The parsing framework is implemented and tested for natural language queries in Air Travel Information System (ATIS) domain.

Chapter 2 introduces some general background on parsing and natural language understanding. It describes LR parsing algorithm – an efficient algorithm, but it cannot deal with ambiguity in natural language system. Then it explains how GLR parser to generate multiple parses from an ambiguous sentence. Then it overviews some natural language systems.

In order to deal with large grammar size for natural language processing, we propose a *grammar partitioning* approach in Chapter 3. We describe the concepts of virtual terminals, and the INPUT/OUTPUT sets which captures the relationships between sub-grammars. Each sub-grammar has its own specialized sub-parser, and we need to combine the sub-parsers by a

technique called *parser composition* in order to provide an overall parse for the input sentence. We describe two approaches for parser composition – (i) composition by cascading and (ii) composition with predictive pruning.

Chapter 4 describes how to integrates sub-parsers during parsing. It introduces the concepts of *Lattice with Multiple Granularity (LMG)*, which is an interface of the sub-parsers and describes how to modify GLR parsing algorithm to adapt a lattice input. Two parser composition algorithms – *parser composition by cascading* and *parser composition with predictive pruning* are implemented. In Chapter 5, experiments are carried out to evaluate different parsing strategy.

Chapter 5 describes parsing experiments on English and Chinese queries in ATIS domain. Our observations include the size of the parsing tables, computational costs, degrees of parse coverage and understanding accuracies. It also presents our semantic interpreter for generating semantic frame. It explores the trade-off between language understanding accuracies and computational costs for the two different parser composition algorithms.

The last chapter summarizes our experimental results and the contributions of this work. We also present our conclusions and future research directions.

Chapter 2

Background

2.1 Introduction

The parsing framework developed in this thesis is based on Tomita's Generalized LR (GLR) parsing algorithm. In this chapter, we first introduced the definitions and notations of context-free grammars (CFG), which is supported by the LR parser. Next, we described the principles of the LR parsing algorithm. Since LR parsers were originally developed for parsing programming languages, it cannot handle an ambiguous grammar for natural language. So we introduce Tomita's GLR parser which can tackle ambiguities. We will also describe some previous work related to complexity analyses for GLR parsing algorithm. It provides a evidence that the GLR parser is an efficient parsing technique.

Parsing technology is embedded in many natural language systems and will review some of these systems, such as GLR*, TINA and PHOENIX in the last section.

2.2 Context-Free Grammars

A grammar is a (finite) mechanism for producing sets of strings (language). A context-free grammar consists rules that have only a single symbol on the left-hand side. Given a context-free grammar rule: $X \rightarrow W$, symbol X can be replaced by W when it appears in a string, no matter the context. A context-sensitive grammar only allows a rule to replace symbol X by W when X occurs between Y and Z in a string: $YXZ \rightarrow YWZ$. Most programming languages are defined by context-free grammar, it is because there are efficient algorithms for parsing these languages, such as LR parsing algorithm. We tied to use LR parser in the thesis, so context-free grammar is chosen.

A context-free grammar (CFG) can be defined as $G = (V_N, V_T, P, Z_0)$. V_N and V_T are the nonterminal and terminal vocabularies of grammar G. P is a finite set of grammar production rules of the form $A \rightarrow \omega$. The V_N set includes all left hand side of any production rule, and V_T set includes all symbols appearing on the right hand side of any production that are not members of V_N. Z_0 is a particular member of V_N, called the starting symbol. The common used notations for symbols in CFG as shown below[1] [2]:

1. Upper-case letters occurring early in the alphabet, such as A, B, C, etc. are used for nonterminal symbols.

2. Lower-case letters occurring early in the alphabet, such as a, b, c, etc. are used for terminal symbols, lower-case.

3. Upper-case letters occurring late in the alphabet, such as X, Y, Z rep-

[1] We will use these notations throughout the remainder of the thesis.

resent grammar symbols, either nonterminals or terminals.

4. Letters occurring late in the alphabet, such as t, u, ..., z are used for strings of terminals.

5. Lower-case Greek occurring letters, α, β, γ, etc. represent strings of grammar symbols.

6. For the production rule A $\rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_k$, $\alpha_1, \alpha_2, ..., \alpha_k$ are the alternatives for A.

7. Unless otherwise stated, the left side of the first production is the start symbol, i.e. a root node of a parse tree by a given grammar.

If $\alpha_1 \rightarrow \alpha_2 \rightarrow ... \rightarrow \alpha_n$ and n$\geq$1, then $\alpha_1 \Rightarrow \alpha_n$. The sequence $\alpha_1, \alpha_2, ..., \alpha_n$ is a *derivation* of α_n from α_1. If each step of a derivation of the form $\alpha At \rightarrow \alpha\omega t$, then the derivation is a *rightmost derivation*, since the rightmost nonterminal symbol is rewritten at each step.

2.3 The LR Parsing Algorithm

In this section, we review the principles of the LR parsing technique [2][3][4]. LR parsers were originally developed for parsing programming languages in the late 1960s and early 1970s [13]. LR(k) parser is an efficient, bottom-up[2] parsing technique that can be used to parse a class of context-free grammars. The "L" stands for left-to-right scanning of the input, the "R" refers to the construction a rightmost derivation in reverse, and the "k" for the number

[2]An LR parser builds the parse tree from the bottom leaves up to the root.

of input symbols of lookahead that is used for parsing. When (k) is omitted, k is assumed to be 1.

It needs an LR parser generator for constructing LR parsing table for a grammar, such as YACC [11][3]. The parsing table is pre-compiled from a given grammar. Throughout this thesis, LR(1) parsing table is used which is implemented in C. Figure 2.1 shows a model of an LR parser. An LR parser consists of a linear stack, a parsing program and a parsing table. The parsing program is the same for all LR parsers; only the parsing table changes from one parser to another. The parsing program reads words from an input buffer one at a time. The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2...X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a symbol called a state. Each state symbol summarizes the information contained in the stack below it, and the combination of the state symbol on the top of the stack and the current input symbol are used to index the parsing table and determine the shift-reduce parsing decision.

The parsing table consists of two parts, an ACTION table and a GOTO table. The GOTO table takes a state s and grammar symbol A as arguments and produces a state, which represents as GOTO$[s, A]$. The ACTION table takes the state s currently on top of the stack and the current input symbol a as arguments, which represents as ACTION$[s, a]$ and produce one of four values:

1. *shift* s, where s is a state. The parser shifts both the current input

[3]YACC is available as a command on the UNIX system, and has been used to help implement hundreds of compilers.

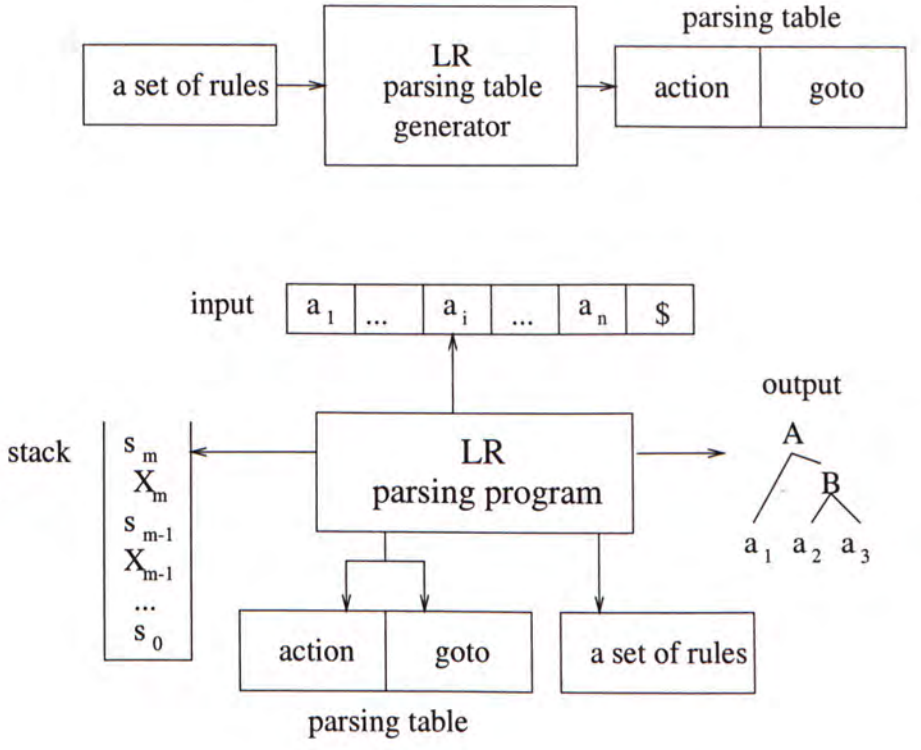

Figure 2.1: A model of an LR parser.

symbol and the next state s onto the stack.

2. *reduce* by a grammar production $A \rightarrow \beta$. Assume s=GOTO$[s_{m-r}, A]$, s_m is the state at the stack top and r is the length of β. The parser first pops $2r$ symbols off the stack, exposing state s_{m-r}. It then pushes both A and s onto the stack. The current input symbol is not changed in a reduce action.

3. *accept*, i.e. parsing is completed.

4. *error*, i.e. the parser discovers an error.

Only a subset of context-free grammars called LR grammars can have an

LR parsing table, which do not have multiple entries. Natural language is inherently ambiguous, whereas programming language are not. Therefore, LR parsing technique cannot be directly used for natural languages. Other algorithms suitable for handling context-free grammars can be used for parsing natural language grammars. In particular, Earley's algorithm [7], GLR algorithm [16][28][30], Chart Parsers [12], and the CKY parsing algorithm [39] were widely used.

2.4 The Generalized LR Parsing Algorithm

Generalized LR (GLR) parsing was introduced by Tomita in 1985 [28]. He proposed a general and efficient method for dealing with action conflicts in the parsing table, this parsing algorithm could handle acyclic context-free grammars. Instead of a linear stack, a *graph-structured stack* (GSS) is used to simulate non-determinism. He also introduced *packed shared parse forest* – a compacted way of representing the possible parse trees for an ambiguous sentence. In this section, we review some major features of GLR and an example in [28][30].

2.4.1 Graph-Structured Stack

A GSS is similar to a linear stack in an LR parser, it contains state nodes and symbol nodes. The state nodes are grouped in a layer field U_i, which is created when parsing terminal in position i in the sentence. In some entries of parsing table, they may contain more than one rule reduction or shift action

if the table is generated from an ambiguous grammar. A GLR parser will perform all rule reductions before shift action. In the following, we describe how it handles multiple reduction and shift actions.

If a stack can be reduced in more than one way, then the top of the stack is made to split to accommodate the various possibilities. Assume the current GSS configuration is displayed in Figure 2.2. The circle represents state node, and square represents symbol node. The English letter from a to d are terminals in GSS. For simplification, layer field U_i and some useless state and symbol nodes are omitted in the following GSS figures. Suppose the stack is to be reduced with each the following three productions in parallel: $F \rightarrow c\ d$, $G \rightarrow c\ d$, and $H \rightarrow b\ c\ d$. After rule reductions, the stack becomes Figure 2.3, where the top of GSS is split into three branches. Notice that F, G and H are the "stack" tops.

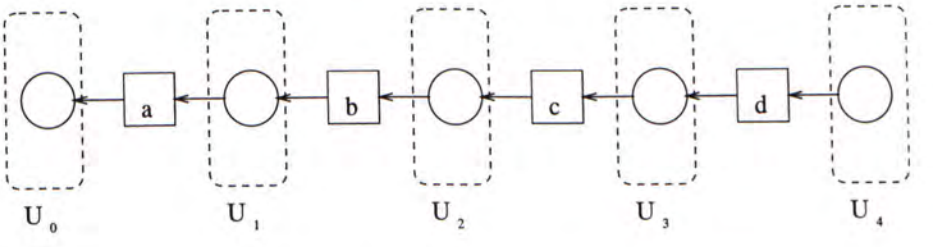

Figure 2.2: The initial stage of a GSS.

If an input symbol needs to be shifted onto more than one stack top, it is done only once by combining the tops of the stack. To continue from the previous example, if i is to be shifted to F, G and H, then the stack becomes Figure 2.4. Observe that i becomes the only stack top.

If two or more branches of the stack turn out to be identical, then they

represent local ambiguity. These branches are merged and treated as a single branch. To continue from the previous example, suppose we are now to reduce the stack by each of the following productions: $J \rightarrow F\ i$,and $J \rightarrow G\ i$. The resultant stack is shown in Figure 2.5. The corresponding packed forest of packed vertex J is displayed in Figure 2.6. We can observe that packed vertex J have two subtrees and say input phrase "c d i" is ambiguous.

2.4.2 Packed Shared Parse Forest

The LR parser can only produce a single parse or sub-parse for a given input. However the GLR parser can produce *multiple* parses. An efficient representation for multiple parse trees is the *packed shared parse forest.*

If two or more trees have a common subtree, the subtree should be represented only once in the parse forest. This is called *subtree sharing* and a parse forest with such property is called a *shared* forest.

Then it can further minimize the representation of the parse forest by *local ambiguity packing*, which works in the following way. The top vertices of subtrees that represent local ambiguity are merged and treated as if there were only one vertex. It is called a *packed vertex*, such as vertex J in Figure 2.5 reduced from GSS in Figure 2.4. A parse forest with both subtree sharing and local ambiguity packing is called a *packed shared forest.*

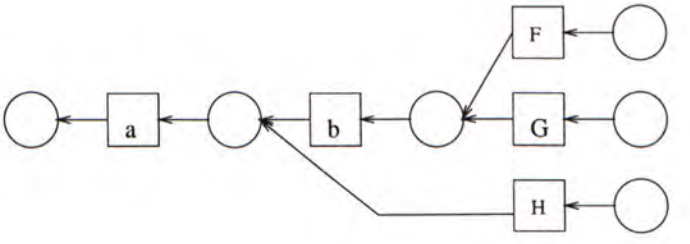

Figure 2.3: The GSS after rule reduction.

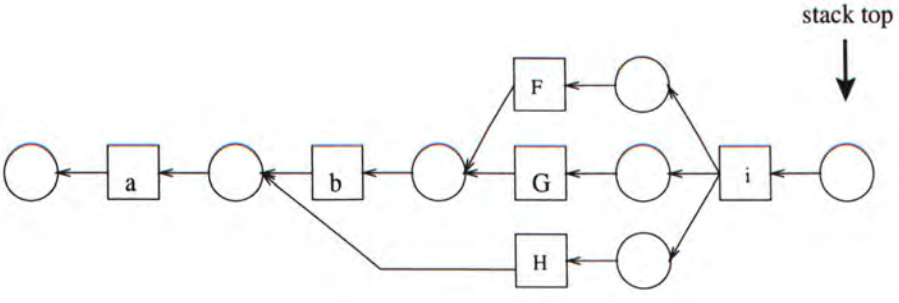

Figure 2.4: The GSS after state shifted.

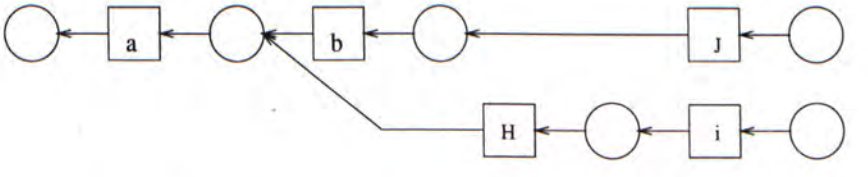

Figure 2.5: The GSS after local ambiguity packing.

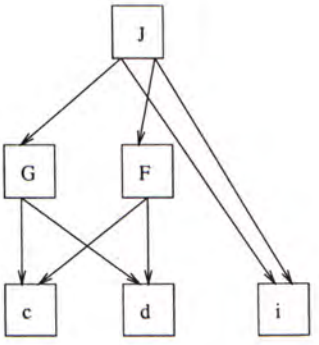

Figure 2.6: The packed forest of parse vertex J after local ambiguity packing.

2.5 Time and Space Complexity

LR parsing parser is a deterministic parser and is highly efficient. The fundamental LR(k) parsing algorithms have a time complexity of $O(n)$, where n is the length of the input.

Tomita [28] showed that parsing time and space complexity of GLR parsing algorithm depends on sentence length, sentence ambiguity and grammar size.

There are some studies to compare GLR parsing algorithm with the other algorithms. Tomita [28] compared the performance of the GLR parser with Earley's algorithm by a similar implementation. GLR parser performs at least two times faster than Earley's algorithm with respect to sentence length, sentence ambiguity and grammar size respectively. Shann [24] compared GLR parser with four different chart parsers: top-down, left-corner, Cocke-Kasami-Younger (CKY) and Bi-directional [26]. The results showed that in the most cases, the GLR parser performed better than other parsers, especially for high ambiguity grammar. Shann attributes the efficiency results to

the parse forest representation.

2.6 Related Work on Parsing

There are a variety parsers which have been proposed for use in natural language systems. This section provides a report of some of these parsers. *GLR** algorithm allows skipping words for robust parsing. The TINA parser integrates ideas from CFG, Augmented Transition Networks (ATN) grammar and the unification concept. The PHOENIX parser spots phrases form the sentence. They are applied to corpora such as Air Travel Information System (ATIS), Resource Management (RM), etc.

2.6.1 *GLR**

Lavie [15][16] proposed *GLR**, a grammar-based parsing system based on Tomita's Generalized LR parsing algorithm, that was designed to be robust to two particular types of extra-grammaticalities, including noise in the input, and limited grammar coverage. *GLR** tries to overcome these types of extra-grammaticalities 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 *GLR** parsing algorithm is designed to find and parse all possible

grammatical substrings of a given input. It uses the same SLR(0)[4] parsing tables that are pre-compiled from the grammar. The difference is the run-time process of the algorithm.

*GLR** skips words of the input string by allowing shift operations to be performed from inactive state nodes in the GSS. Since the parser is LR(0), reduce operations need not be repeated for skipped words. Information about skipped words is maintained in the symbol nodes that represent parse sub-trees.

The *GLR** parsing algorithm constructs parses for all parsable subsets of a given input sentence. To find the "best" parse from the set of parses returned by the parser, a scoring procedure that ranks each of the parse found is used. The scoring procedure takes into account the number of words skipped and the fragmentation of the parse. Both measures are weighed equally.

In order to guarantee a reasonable running time, a heuristic is developed and added to the parser to prune parsing options that are unlikely to produce a maximal parse. The heuristic limits the number of inactive state nodes from which a input symbol is shifted.

2.6.2 TINA

TINA [22][23] is a natural language system, which was developed for spoken language applications. It combines key ideas from context-free grammars, Augmented Transition Networks (ATN's) and the unification concept.

[4]SLR(0) stands for simple LR without lookahead input symbol, its parsing table is the easiest one to implement compared with all the other LR parsing tables.

TINA is based on a context-free grammar augmented with a set of features used to enforce syntactic and semantic constraints. The grammar rules represent syntactic structure at the high levels of parse tree and represent semantic structure at the low levels. All meaning-carrying content of the sentence is encoded in the names of categories of the parse tree. The grammar is converted to a network structure by merging common elements on the right-hand side (RHS) of all rules sharing the same left-hand side (LHS) category. Probabilities on all arcs in the network are assigned automatically from a set of example sentences.

The parser consists of a stack decoding search strategy, with a top-down control flow, and includes a feature-passing mechanism to deal with long-distance movement, agreement, and semantic constraints. The stack is similar to the stack in LR parsing algorithm, it performs one of four actions: shift, reduce, accept and reject. At any time, a distinguished subset of active parse nodes are arranged on a priority queue. Parse nodes are placed on queue prioritized by probability. The parsing process can terminate on the first successful completion of a sentence, or the nth successful completion if more than one hypothesis is desired.

2.6.3 PHOENIX

The CMU PHOENIX system [32][33][34] is developed for understanding spontaneous speech, is implemented for the Air Travel Information Systems (ATIS) task. The speech recognition and language parsing are two components of the system. The recognizer produces a single best word string from the speech

input, and then the word string is passed to the frame-based parser. The PHOENIX parser can accommodate extra-grammatical text input.

The concept of flexible parsing combines frames-based semantics with a semantic phrase grammar. Semantic information is represented in a set of frames. Each frame contains a set of slots representing a pieces of information. A partitioned semantic phrase grammar is used to fill the slots in the frames. Each slot type is represented by a separate finite-state network which specifies different ways of saying the meaning represented by the slot. The non-terminals of the grammar rules are semantic concepts, instead of parts of speech. An input which do not form a grammatical English sentence are still parsed. The grammar is compiled into a set of finite-state networks. Many small networks instead of one large network for partitioned grammar, that significantly reduces the overall size of the system. Networks are top-down and left-to-right oriented.

The operation of the parser resembles "phrase spotting". A beam of possible interpretations are generated. An interpretation is a frame with some filled slots. The finite-state networks are used to perform pattern matches against input word strings. When a slot matches, it extend all active frames that contain that slot and activate any currently inactive frames that contain the slot.

When a single phrase can be parsed with two or more frames, leading to multiple interpretations, then the one with the lower score is pruned. The score for an interpretation is the number of input words that it accounts for. The best scoring interpretation becomes the system's output. In addition,

slots have be different levels of hierarchy. Higher levels slots can contain several lower level slots. When two interpretations have the same score, higher level slots are preferred to lower level ones since the associations between concepts are more tightly bound.

2.7 Chapter Summary

In this chapter, we described some relevant background information for this thesis. We first reviewed the definitions of context-free grammars, which are used by LR parsers and many other parsers. Then we introduced an LR parser, which is efficient but cannot handle ambiguity. The GLR parser is modified from LR parser was also presented, that can handle ambiguity. In addition, we discussed time and space complexity analyses for GLR parsing algorithm. Finally, we introduced some recent parsers for natural language understanding. In the next two chapters, we will describe in detail how to embed the GLR parser in our parsing framework.

Chapter 3

Grammar Partitioning

3.1 Introduction

This chapter discusses in detail the method of grammar partitioning. The main objectives of grammar partitioning are to reduce the size of parsing table for a large grammar, and to obtain a modular parser in natural language processing, as described in Section 3.2. We review previous work on grammar partitioning in Section 3.3. Then we present the features of our grammar partitioning approach, including the concepts of *virtual terminals*, INPUT and OUTPUT non-terminal sets. In the last section, we show an example to illustrate how to partition a grammar into several smaller sub-grammars.

3.2 Motivation

Natural language is inherently ambiguous. There are many ways to express the same meaning. Therefore, to understand natural language we may need

to handcraft a lot of grammar rules, even for a restricted domain. The LR parsing algorithm is efficient and widely used. However, it is not practical for large grammars. In order to obtain an efficient and modular parser, we propose a grammar partitioning approach.

Since CFGs are the backbones for many natural language processing systems, such as [9][23], we will focus on CFGs. Earley [7] and Ukkonen [31] proved that the following artificial grammar G_n is exponential for LR(k) parsers, and even for any right parsers respectively:

$S \rightarrow A_i (1 \leq i \leq n)$

$A_i \rightarrow a_j A_i (1 \leq i \neq j \leq n)$

$A_i \rightarrow a_j B_i \mid b_i (1 \leq i \leq n)$

$B_i \rightarrow a_j B_i \mid b_i (1 \leq i, j \leq n)$

These results present a potential problem for LR(k) parsers and their applications to natural language processing. Korenjak [14] proved that grammar partitioning and parsing table regeneration for partitioned grammars significantly reduces the computation required to generate an LR(1) parsing table for a large grammar. Weng and Stolcke [36] provided a more flexible scheme for grammar partitioning and parser composition. He proposed to partition a grammar into subsets based on production rules and compose sub-parsers of different types. As an example, he proposed an algorithm based on Tomita's GLR parsers and composed in strict top-down manner.

From a practical viewpoint, sublanguages in a large parser may be described by pre-existing sub-grammars using specialized parsing algorithms. Implementation and maintenance constraints may prevent integration of the

sublanguage modules into a monolithic system. A framework for partitioning grammars and composing parsers of different types would be useful for the modularity of a parsing system. If the parsing tables can be constructed separately, then if any single sub-grammar needs to be modified, we only need to regenerate the parsing table to the modified sub-grammar.

3.3 Previous Work on Grammar Partitioning

Korenjak developed a practical method for constructing LR(k) processors in [14]. His technique is based on the original LR(k) processors described by Knuth [13]. It involves partitioning the given grammar into a number of smaller parts to reduce the effort required to construct the LR(k) parsing table as well as the overall sizes of the resulting parsing tables.

LR(k) grammar is a context-free grammar G, which can be defined as $G = (V_N, V_T, P, Z_0)$. Korenjak mainly considered LR(1) grammars. His approach is to partition the grammar based on non-terminals. The following describes the general scheme for grammar partitioning. Let $G = (V_N, V_T, P, Z_0)$ be a large grammar for constructing an LR(1) processor. Choose Z_1, ..., Z_n be distinct nonterminal symbols other than Z_0 in V_N, and let $\overline{z}_1, ..., \overline{z}_n$ be new terminal symbols not in V_N or V_T. Rules in P are from 1 to π. Let $G' = (V_N', V_T', P', Z_0')$ be the 1-augmented grammar[1] associated with G, and assign the production $Z_0' \rightarrow Z_0\#$[2] the number 0. For $0 \leq$ i $\leq$ n,

[1] $G' = (V_N', V_T', P', Z_0')$ is called the *k-augmented grammar* associated with $G = (V_N, V_T, P, Z_0)$ for fixed k ≥ 0 and Z_0', $\# \notin V$, if $V_N' = V_N \cup \{Z_0'\}$, $V_T' = V_T \cup \{\#\}$ and $P' = P \cup \{Z_0' \rightarrow Z_0\#^k\}$. Number the productions of G' as in G, assigning $Z_0 \rightarrow Z\#^k$ the number 0.

[2] The symbol # is called the endmarker, which is same as "$" symbol in [2].

let $G_i = (V_{N_i}, V_{T_i}, P_i, Z_i)$, be the reduced form[3] of $(V_N, \overline{V}_T, P_i, Z_i)$, where $\overline{V}_T = V_T \cup \{\overline{z}_j \mid 0 \leq j \leq n\}$ and $P_i = \{A \rightarrow \overline{\alpha} \mid A \rightarrow \alpha \in P$ and $\overline{\alpha}$ is obtained from α by replacing every occurrence of Z_j $(j \neq i)$ by $\overline{z}_j\}$. Assign the rule $A \rightarrow \overline{\alpha}$ in P_i the number that $A \rightarrow \alpha$ has in P. For each i, let $G_i' = (V_{N_i}', V_{T_i}', P_i', Z_i')$ be the 1-augmented grammar associated with G_i, with endmarker $\#_i$. We can say the grammar G is partitioned into G_i for $0 \leq i \leq n$.

After an LR(k) processor is constructed for every subgrammar by Knuth's algorithm, Korenjak propose an algorithm for constructing an LR(k) processor for the entire grammar from them, which is also contained Knuth's ideas for decreasing the computation time for an LR(k) table. Obtaining a parsing table using this partitioning scheme is a trial-and-error process, since a partition must be chosen and the corresponding table generated. If a particular partition fails to produce an LR(1) parsing table (the parsing table contains multiple entries), the entire process may have to be repeated.

Korenjak's grammar partitioning scheme had been used for generating programming language parsers, such as ALGOL, BASIC and CDL1. However, this scheme is not suitable for partitioning grammar for natural language, since it cannot handle non-LR(1) grammars.[4] Due to poor partition choices, the constructing table algorithm may fail even for LR(1) grammars.

[3] A grammar is in reduced form if all "useless" nonterminals are removed. For each $A \in V_N$, $\exists \alpha, \beta, t$ such that $S \Rightarrow \alpha A \beta \Rightarrow \alpha t \beta$.

[4] The generated parsing table contains multiple entities if non-LR(1) grammar is used.

3.4 Our Grammar Partitioning Approach

3.4.1 Definitions and Concepts

Similar to CMU PHOENIX parser, we partition the entire grammar into a set of sub-grammars. Instead of finite-state networks, we compile the sub-grammars into a set of LR(1) parsing tables in order to reduce the overall size of the parsing table for the entire grammar.

Basically, we use Korenjak's partitioning scheme and Weng and Stolcke's *calling graph* concept [36] for our approach. We partition a grammar into subsets based on non-terminals, and create sub-grammars for the subsets accordingly. Suppose the original entire CFG is defined as $G = (V_N, V_T, P, S)$, where S is the start symbol. In the remainder of the thesis, we will use the following definitions for grammar partitioning:

Definition 1: $\{G_i = (V_{N_i}, V_{T_i}, P_i, Z_i)\}$ is called a partition of G, where i is an integer $0 \leq i \leq n$. G is partitioned into n sub-grammars, according to Korenjak's partitioning scheme.

Definition 2: *Virtual terminal* is prefiexed with *vt*. It essentially a non-terminal, but acts as if it were a terminal. vtZ is same as $\overline{z}$ in Korenjak's definition.

Definition 3: *Virtual terminal* in V_{T_i} is called the *input* of G_i, $INPUT_{G_i}$ and Z_i is called the *output* of G_i, $OUTPUT_{G_i}$.

Definition 4: $\{G_0 = (V_{N_0}, V_{T_0}, P_0, S)\}$ is called the master sub-grammar of G. $\{G_i = (V_{N_i}, V_{T_i}, P_i, Z_i)\}$ for $1 \leq i \leq n$ is called the slave sub-grammar of G.

The interaction among different sub-grammars is through non-terminal sets, i.e., INPUT and OUTPUT, and a virtual terminal technique. For a sub-grammar, its INPUT is a set of virtual terminals that were previously parsed by other sub-grammars. To be more precise, these virtual terminals vtA are on the right-hand side (RHS) of some rules in this sub-grammar and non-terminals A are on the left-hand side (LHS) of certain rules in some other sub-grammar(s). The OUTPUT of a sub-grammar are those non-terminals A that were parsed based on this sub-grammar and their virtual terminal vtA used by other sub-grammars as their INPUT symbols. To be more precise, those non-terminals A are on the LHS of some rules in this sub-grammar and their vtA are INPUT set members of some other subset(s). In other words,

we may view a partitioned subset of production rules of a grammar as a one-value function, it takes virtual terminals in INPUT as its input, and returns a non-terminal in OUTPUT as its output. A directed *calling graph* for the sub-grammar set of G is then defined as (V, E), where V is the sub-grammar set and $E = \{(C, B)\}$, with the overlap of the OUTPUT of B and the INPUT of C being nonempty, where C and B are the sub-grammars in V. It can be proven that the partitioned grammar will lead to the same recognizable language.

Each partitioned sub-grammar is assigned with *level index* (*ID*). The level ordering criteria are: a sub-grammar is assigned to level i if its input are virtual terminals with level $< i$. The master sub-grammar is assigned with the highest level index. The lowest level index should be zero. If the calling graph of the sub-grammar set contains cycle, the involved sub-grammars are assigned with the same indices. This level index is used for composing sub-parsers and will be described in the next chapter.

The difference between our approach and Korenjak's approach is that we use a GLR parser to handle each partitioned sub-grammar. We consider the problem of grammars other than LR(1) grammar. Besides, we also use a flexible parser composition algorithm to integrate sub-parsers instead of using a trail-and-error LR(1) table regeneration algorithm for the entire grammar. We will describe it in detail in the next chapter.

3.4.2 Guidelines for Grammar Partitioning

Korenjak [14] provided some useful guidelines for grammar partitioning. As an initial step towards automatic grammar partitioning, Weng et al. [35] also proposed a few guidelines, including merge and split operations based on the statistical and non-statistical characteristics of the calling graph. Due to the lack of training data, we use a non-statistical guidelines for grammar partitioning. We summarize the guidelines as below:

1. Each partitioned grammar G_i can have as many rules as possible, as long as the generated parsing table is within an affordable size.

2. Partition the overall grammar G into G_i for i=0,1, ..., n such that there are frequent interactions within G_i, but fewer interactions between G_i. This will reduce the complexity of composing parser for highly ambiguous grammars.

3. Select a non-terminal as OUTPUT of a G_i which appears in many different parts of grammar. This will help to reduce duplicate grammar rules after partitioning.

Actually, there is no unique way to partition a grammar. We try to use the above guidelines to reduce total size of parsing table and complexity of composing sub-parsers.

3.5 An Example

In this section, we are using the following toy grammar G_S[5] to illustrate how to partition grammar into several sub-grammars:

G_S:
1. S→NP VP
2. S→S PP
3. NP→n
4. NP→det n
5. NP→NP PP
6. PP→prep NP
7. VP→v NP

From the above grammar, S denotes sentence rule (start symbol), NP denotes noun phrase, VP denotes verb phrase, PP denotes prepositional phrase, n denotes noun, *det* denotes determiner, *prep* denotes preposition and v denotes verb. In general, the terminals of the grammar are in lower-case, the non-terminals are in upper-case, and the virtual terminals are prefixed with *vt*. The parsing table of G_S is shown in Table 3.1. In the ACTION table, *sh s* means shift state *s*, *re n* means reduce by a grammar production *n* and *acc* means accept. In the GOTO table, the number *s* means goto state *s*. Besides, empty entity means error.

We select S, NP and PP non-terminals for partitioning. Then the original G_S is partitioned into G_0, G_1 and G_2 as shown in below, their respective parsing tables are shown in Table 3.2, 3.3 and 3.4. G_0 is the master grammar, and G_1 and G_2 are the slave grammars. The calling graph of these

[5]This grammar is taken from Tomita's book [28].

State	Action					Goto			
	n	det	prep	v	$	S	NP	VP	PP
0	sh 3	sh 4				1	2		
1			sh 6		acc				5
2			sh 10	sh 9				7	8
3			re 3	re 3					
4	sh 11								
5			re 2		re 2				
6	sh 13	sh 14					12		
7			re 1		re 1				
8			re 5	re5					
9	sh 13	sh 14					15		
10	sh 3	sh 4					16		
11			re 4	re 4					
12			sh 6,re 6		re 6				17
13			re 3		re 3				
14	sh 18								
15			sh 6,re 7		re 7				17
16			sh 10,re 6	re 6					8
17			re 5		re 5				
18			re 4		re 4				

Table 3.1: LR(1) parsing table for unpartitioned grammar G_S.

three sub-grammars are shown in Figure 3.1, which shows their input and output relationship. The INPUT of G_0 are $vtNP$ and $vtPP$, the OUTPUT of G_1 and G_2 are PP and NP respectively. Therefore, the directed edges E from G_0 to G_1 and G_2 are added to the calling graph. In the same way, there are two edges between G_1 and G_0. The level indices are assigned according to our level ordering criteria. We can observe that parsing table of G_S has more states than sum of states of these three partitioned sub-grammars. The parsing table of G_S has 19 states, and partitioned grammars has totally 7+6+4=17 states.

State	Action				Goto	
	vtNP	vtPP	v	$	S	VP
0	sh 2				1	
1		sh 3		acc		
2			sh 5			4
3		re 2		re 2		
4		re 1		re 1		
5	sh 6					
6		re 3		re 3		

Table 3.2: LR(1) parsing table for partitioned grammar G_0.

G_0:
INPUT = {vtNP, vtPP}
OUTPUT = {S}
1. S→vtNP VP
2. S→S vtPP
3. VP→v vtNP

G_1:
INPUT = {vtPP}
OUTPUT = {NP}
1. NP→n
2. NP→det n
3. NP→NP vtPP

G_2:
INPUT = {vtNP}
OUTPUT = {PP}
1. PP→prep vtNP

State	Action				Goto
	n	det	vtPP	$	NP
0	sh 2	sh 3			1
1			sh 4	acc	
2			re 1	re 1	
3	sh 5				
4			re 3	re 3	
5			re 2	re 2	

Table 3.3: LR(1) parsing table for partitioned grammar G_1.

State	Action			Goto
	prep	vtNP	$	PP
0	sh 2			1
1			acc	
2		sh 3		
3			re 1	

Table 3.4: LR(1) parsing table for partitioned grammar G_2.

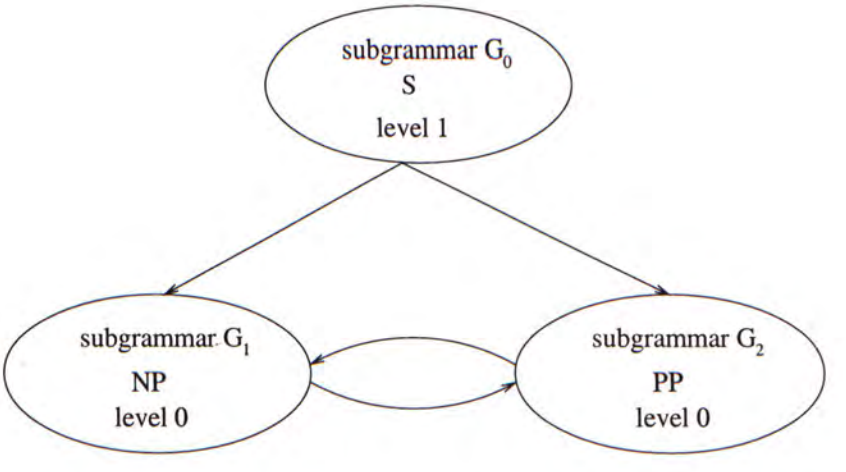

Figure 3.1: The calling graph of the sub-grammars G_0, G_1 and G_2.

3.6 Chapter Summary

In this chapter, we first motivated the need for grammar partitioning. We then described related previous work and presented our grammar partitioning approach in terms of definitions, concepts and guidelines. Finally, we showed a grammar partitioning example, and demonstrated how partitioning can reduce the total size of the parsing tables. In the next chapter, we will describe how to compose sub-parsers for partitioned sub-grammars.

Chapter 4

Parser Composition

4.1 Introduction

As mentioned previously, a grammar is partitioned into several sub-grammars, each operates with its own specialized sub-parser. We need to compose all sub-parsers in order to obtain an overall parse of the input. Each sub-parser is a GLR parser and contains an LR(1) parsing table. A lattice is introduced to serve as an interface among different sub-parsers. Each sub-parser follows GLR lattice parsing algorithm which will be described in Section 4.2. We also present two approaches towards parser composition. The first one is the *parser composition by cascading*, which describes in Section 4.3.1. The second one is the *parser composition with predictive pruning*, which will be described in Section 4.3.2. At the end of this chapter, we also compare these two composition algorithms.

4.2 GLR Lattice Parsing

4.2.1 Lattice with Multiple Granularity

The input to our parser is a string of words. With grammar partitioning of sub-parsers, our parsing framework needs a medium to communicate between sub-parsers. In the other words, it requires an interface to record the INPUT and OUTPUT of all sub-parsers. We introduce a lattice representation to play this role. Our lattice is a directed acyclic graph (DAG), which is a set of virtual terminals, terminals and transitions. All virtual terminals and terminals are nodes in the lattice and connected by transitions. It is an efficient way to represent a set of (virtual) terminal nodes. Each node is assigned a name and an index. There is a sentence start node <s> and a sentence end node '$'. The sentence START is assigned the index 0, and the sentence END is assigned the index n+1, when n is the sentence length. The terminals are indexed according to their position in the sentence. Virtual terminals are indexed with the last positions they cover in the lattice. Upon parsing, an input sentence is converted into a lattice. For example, the input sentence is "i saw a man with a telescope", i.e. "n v det n prep det n" is converted to a lattice as shown below:

<s> 0→n 1→v 2→det 3→n 4→prep 5→det 6→n 7→$ 8

The number in each lattice node indicates its index. Assume there is a virtual terminal node $vtNP$ and it covers two terminal nodes *det* with index 3 and n with index 4, then $vtNP$ is assigned with index 4. The lattice is shown in Figure 4.1

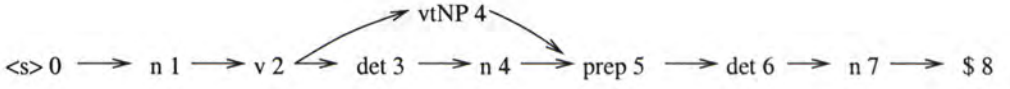

Figure 4.1: An example of a lattice.

During parsing, both terminals and virtual terminals are represented in the same lattice, but nonterminals that are not the output of any sub-grammar cannot be placed there. This implies that only the granularity represented by sub-grammars is present in the lattice. Therefore, the structure is termed *lattice with multiple granularity (LMG)*. This representation provides a good way for book-keeping during parsing.

4.2.2 Modifications to the GLR Parsing Algorithm

Each partitioned sub-grammar has a GLR sub-parser. However, our parser cannot handle CFGs with cycles[1] or ϵ productions. A grammar is *cyclic*, if there exists a nonterminal which can be reduced to itself. To handle the LMG, we modify our GLR parser in a way similar to that proposed in [29]. Parsing strictly follows the topological order of the lattice nodes, i.e. a lattice node will be parsed only after all its previous nodes (i.e. left neighbors) are parsed. Basically, there are two modifications to the original string version GLR algorithm:

[1]Tomita's original version of GLR algorithm cannot to handle infinitely ambiguous and cyclic grammars. Farshi [19] proposed a modified version of GLR algorithm to handle these cases.

- **Modification 1: Active and Inactive States Nodes**

As mentioned in Section 2.4.1, our GLR parsers use a graph-structured stack (GSS). The GSS contains a group of layer fields U_i. As shown in Figure 4.2, U_4 is at the top of the GSS, and U_0 is at the bottom, i.e. U_4 is the "stack" top, and U_0 is the "stack" bottom. If we let U_n be the "stack" top in a GSS, and U_m be the layer field immediately underneath, then n must be greater than m. If the GLR parser only processes input in the form of a string, then $n = m+1$. Furthermore, as we move from the "stack" top to the "stack" bottom, the sequence of layer fields is arranged as $U_n, U_{n-1}, U_{n-2}...U_0$. However, if the GLR parser needs to process input in the form of a lattice, then $n > m$ but $n = m + 1$ may *not* necessarily be true. In other words, as we move from the "stack" top to the "stack" bottom, our sequence of layer fields may not occur in consecutive order.

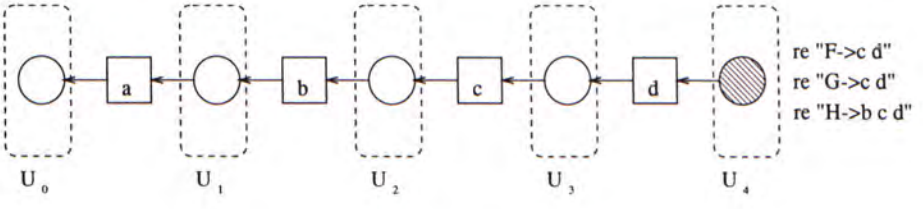

Figure 4.2: The initial stage of a GSS.

In Figure 4.2, we only have a single *active* state at the stack top U_4. Going from Figure 4.2 to 4.3, since we are using an LR(1) parsing table, we read in the next input symbol i, and find three possible reductions according to the rules: $F \rightarrow c\ d$, $G \rightarrow c\ d$, and $H \rightarrow b\ c\ d$. Upon reduction, the node in U_4 of Figure 4.2 is *inactivated*, and our reductions produce three new *active*

nodes in U_4 of Figure 4.3. Here, only the active nodes are shaded for the layer field U_4.

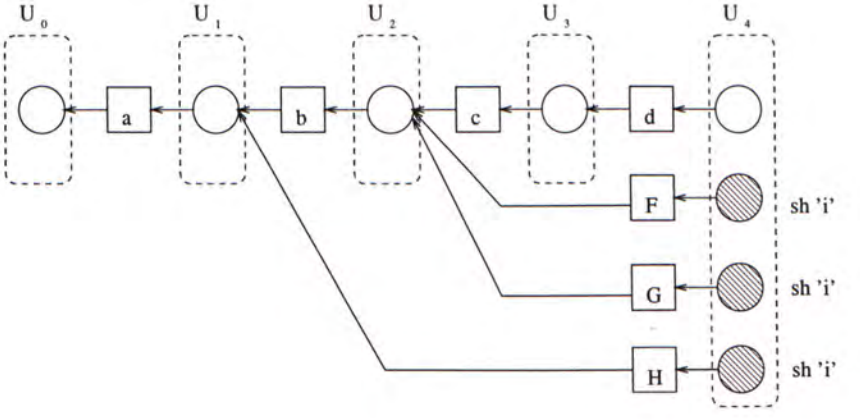

Figure 4.3: The GSS after rule reductions.

Proceeding to Figure 4.4, the GLR parser then reads in the input symbol i, checks against each of the three active nodes, and finds that i can be shifted onto the GSS for all three nodes. Hence i is pushed to the stack top as a *single active state.* By now, the three states have become *inactive*, and we are left with a single active state (shaded) at the stack top in Figure 4.4.

Suppose the lattice (LMG) input we are processing is shown in Figure 4.5. Since we have just processed the input symbol i (with index=5 as seen in Figure 4.5), according to the topological order, we will be processing the input symbol k (also with index=5) as our next step. The LMG shows that the left neighbor of k is d, which has index=4. Hence we refer back to the layer field U_4 in the GSS (Figure 4.4). The state node connected to the input symbol d in U_4 is hence *reactivated*, and we refer to the parsing table to process k. Thereafter we proceed back to the LMG (Figure 4.5) and read

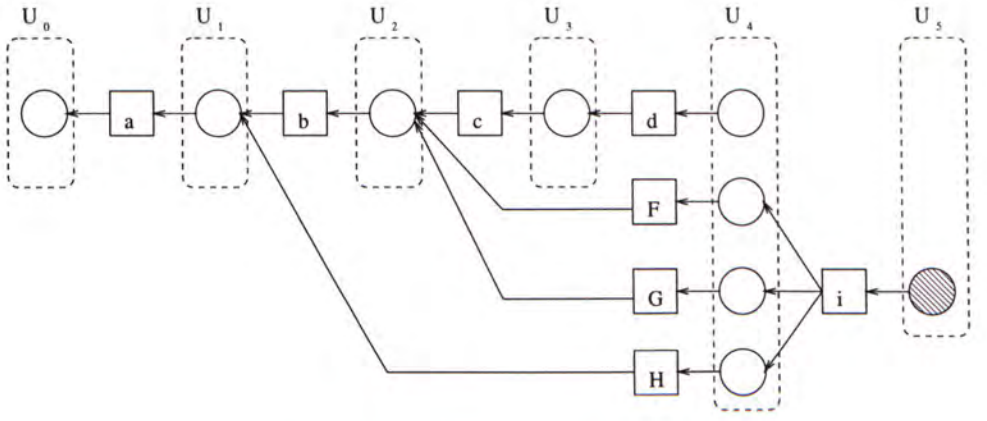

Figure 4.4: The GSS after shift i.

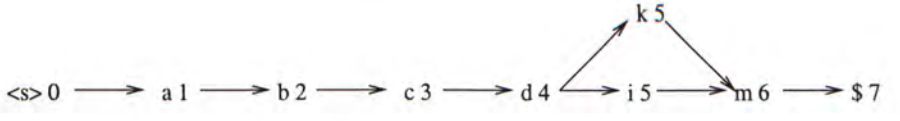

Figure 4.5: The input LMG for example in Modification 1 .

the next input symbol, i.e. m with index=6.

- **Modification 2: Inserting a Robust End for Partial Parses**

Sub-parsers need to decide where to end a partial parse (or sub-tree) according to their sub-grammars. A partial parse can end within a sentence. A straightforward implementation is to consider the alternative of a robust END symbol '$' whenever we read in a new input symbol (terminal or virtual terminal) from the LMG input. In practice, when the sub-parser reads in a new input symbol from the LMG, the sub-parser refers to the symbol's left-neighbor in order to figure out where to insert the new symbol in the GSS. At this point, we have modified our sub-parser such that it also considers the

insertion of a robust END '$'.

- **An Example**

This section traces through the lattice parsing algorithm with the partitioned grammar shown in Section 3.5. Corresponding parsing tables are also given in that section. The input sentence is "n v det n prep det n". Suppose the input sentence becomes an LMG as in Figure 4.6 after several parsing operations.

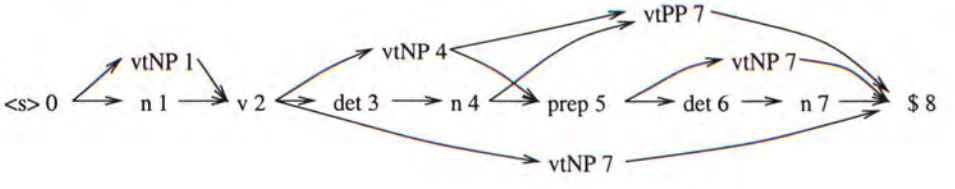

Figure 4.6: The initial stage of the LMG.

Now, suppose sub-parser of G_0 starts at $vtNP$ node with index 1, which is connected to the start symbol <s>. G_0 will create a virtual terminal with label vtS if it parses successfully. Figure 4.7 shows the first several steps of parser operation in GSS. As mentioned in Section 2.4, a circle represents a state node, a square represents a grammar input symbol node and U_i represents the layer field of GSS. A shaded state node represents an active node, while a blank state node represents an inactive node. Similar to the regular GLR algorithm, the GSS only contains one active state 0 node at the beginning. The parser reads the first input symbol $vtNP$ and a robust END '$' together. Since state 0 with '$' produces an error according the parsing table of G_0 in Table 3.2, it only performs *sh 2* – shift $vtNP$ together with

state 2 onto the GSS as shown in the first diagram in Figure 4.7. The next input v can connected to $vtNP$ in GSS, so the parser reads it in the next step as displayed in second diagram in Figure 4.7. From the input lattice in Figure 4.6, there are three next connected node *det* with index 3, $vtNP$ with index 4 and 7 can be proceeded in the following step. Since *det* does not belong to the terminal or nonterminal sets of G_0, it cannot be processed by G_0 parser. Then the parser reads $vtNP$ with index 4 first as shown in the third diagram in Figure 4.7. Since $vtNP$ is assigned with index 4, U_4 is created when it is pushed onto GSS, as in the fourth diagram in Figure 4.7. We can observe that U_3 is skipped. However, U_i is still keep in descending order from the top of the GSS. Active state 6 with $vtPP$ and '$' produce *re 3* – reduce rule $VP \rightarrow v\ vtPP$ in the fourth diagram in Figure 4.7. VP and state 4 are created in U_4 in the fifth diagram in the Figure 4.7. After rule 1 reduction ($S \rightarrow vtNP\ VP$), the GSS becomes the sixth diagram in the Figure 4.7. State 4 with '$' produces *acc* – the phrase "n v det n" can be successfully parsed. Then it can create a virtual terminal node with label vtS onto the lattice. Therefore, sub-parser is not restricted to end at the sentence end, it can be ended anywhere if it can.

After some similar parser operations, the GSS becomes Figure 4.8. Before the parser reads the sentence END symbol '$', it should read $vtNP$ with index 7 first as it needs to parses the input node according to topological order. This $vtNP$ node's previous connected node is v with index 2. The parser *reactivates* state 5 node in U_2, which is predecessor of v in GSS. Then the parser reads $vtNP$ together with '$' in the same manner as before. The final

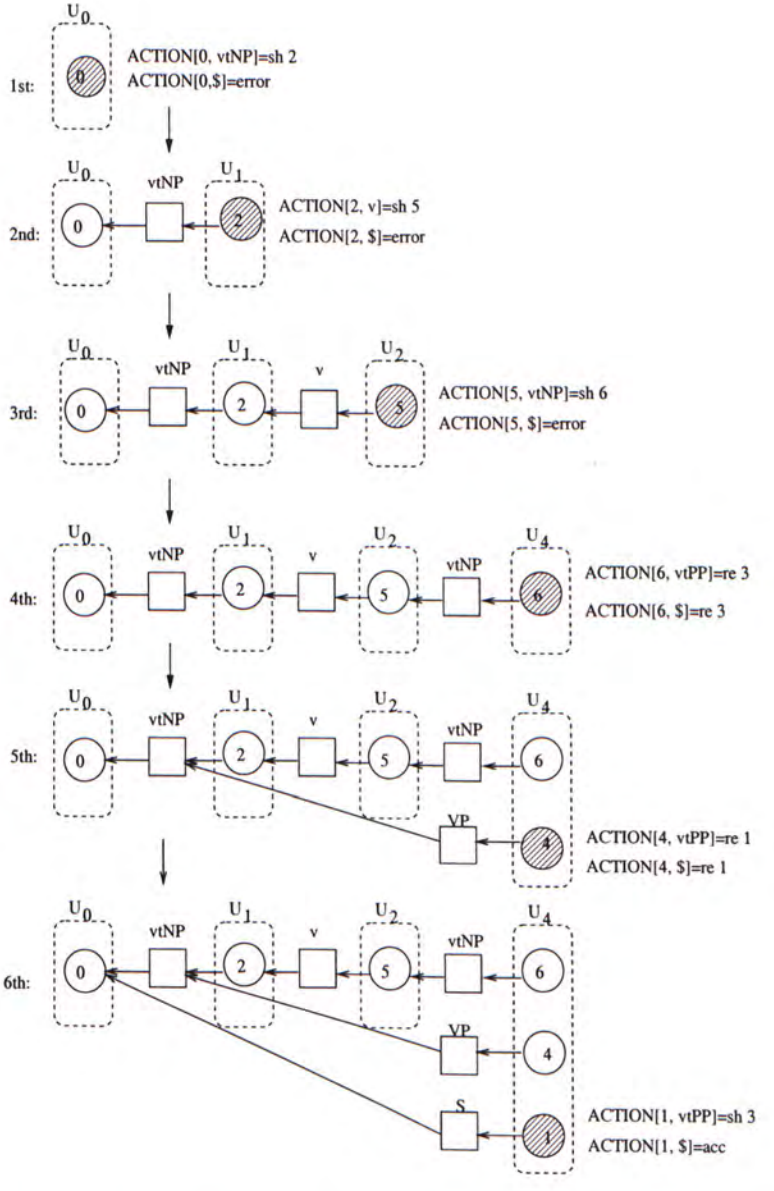

Figure 4.7: Beginning steps of parser operations in the GSS.

stage of GSS is shown in Figure 4.9. The parser can successfully parse till the end of the sentence. The resultant parse forest is displayed in Figure 4.10. Notice that the five virtual terminals in the parse forest are the same with those in the input lattice in Figure 4.6. Including the previous created virtual

terminal vtS, the parser can created two virtual terminals onto the LMG and the resultant parse forests are attached on the virtual terminals.

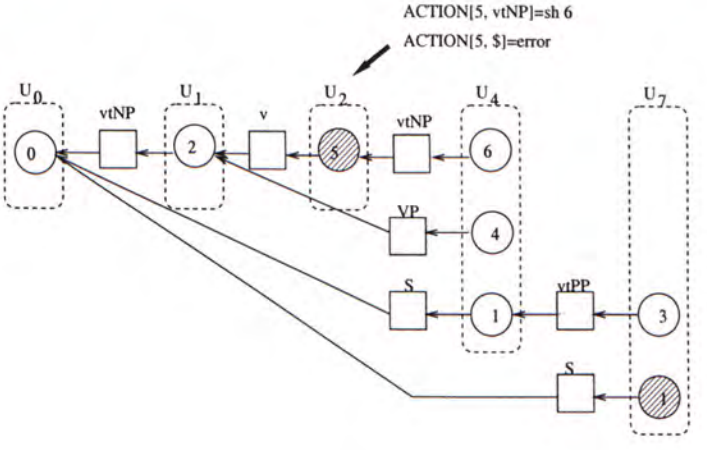

Figure 4.8: The trace of parser in GSS when it reads $vtNP$ with index 7.

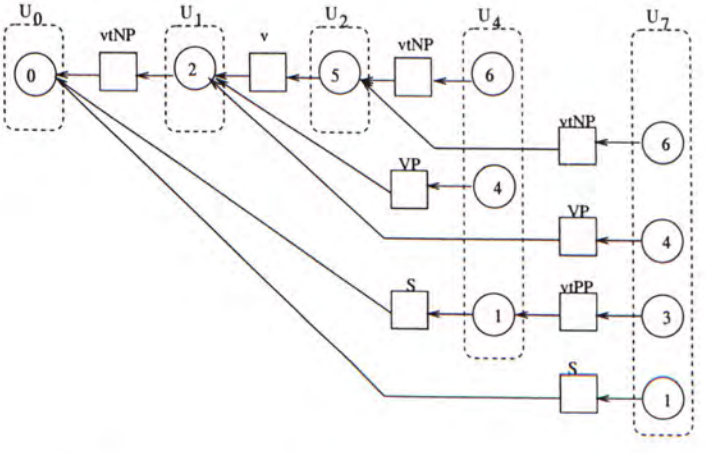

Figure 4.9: The final stage of the GSS.

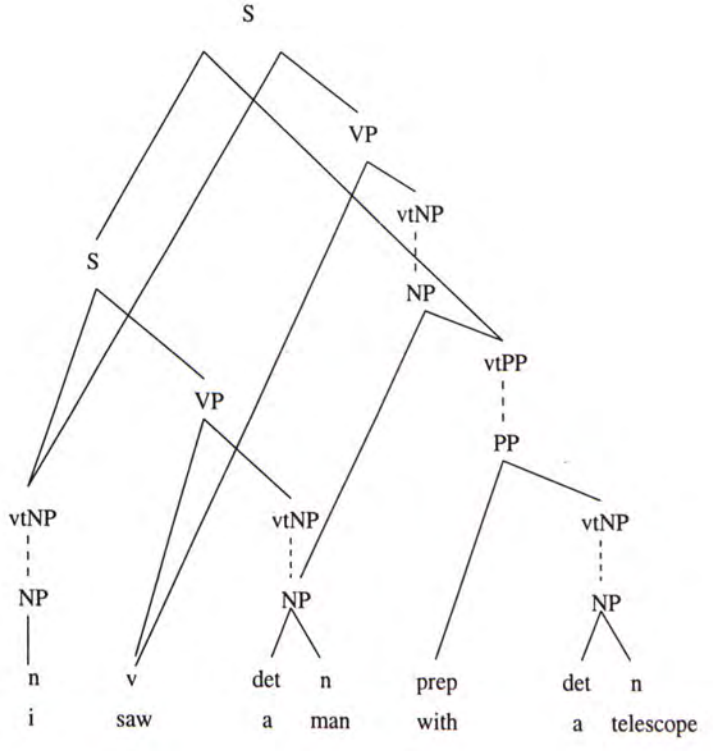

Figure 4.10: The resultant parse forest from sub-parser of G_0.

4.3 Parser Composition Algorithms

After we introduced how each sub-parser parses an LMG input, we will describe how all sub-parsers cooperate to parse the input sentence into a parse forest. The parse forest should be same as the one produced by one parser for the overall grammar. We propose two parser composition algorithms – *parser composition by cascading* and *parser composition with predictive pruning*. There are some common aspects for these two composing algorithms:

1. Each sub-parser is a GLR parser capable of lattice parsing;

2. The GSS and the resultant shared-packed forest are private to each sub-parser;

3. The LMG is shared by all sub-parsers for input and output during parsing.

4.3.1 Parser Composition by Cascading

Parser composition by cascading is straight-forward and relatively easy to implement. The indices of lattice nodes is used to determine the parsing order of each sub-parser. For two integers i and j where $i < j$, each sub-parser will parse a lattice node with index i before another one with index j. This makes sub-parser parses the input node in topological order in the lattice. The level indices of sub-grammars, which defined in Section 3.4, are used to determine the sequence of invocation of sub-parsers. Figure 4.11 shows the control flow of cascading.

There is a stack θ which stores all initial terminal nodes in the input sentence. The first node in the sentence is pushed onto the stack first and the last node in the sentence becomes the top of the stack θ. During parsing, newly created virtual terminals are dynamically added to the stack θ. This stack θ acts as an input list. The sub-parsers in the same level start to parse at the (virtual) terminal on the top of stack θ, and until it reaches the last (virtual) terminals in the stack θ. As a result, a sub-parser can read virtual terminals which have the same level index. This characteristic is important if there is a cycle in the calling graph of sub-grammars.

The mechanism take a query sentence and converts it into an LMG. The sub-parsers at the lowest level are activated to parse the LMG, and leave their corresponding virtual terminals on the LMG when they parse successfully. If

For each level i from the lowest level (0) to the highest level (integer N):
For each node w in the lattice:
For each $parser_j$ in level i:

- if the $parser_j$ can be successfully parsed and return a parse tree with root node labeled as $OUTPUT_j$:
  - if there does not exists a virtual terminal with label $vtOUTPUT_j$ which covers the (virtual) terminal(s) as the newly created node
    * create a virtual terminal label as $vtOUTPUT_j$ with root node $OUTPUT_j$
    * add the virtual terminal onto the lattice by linking the transitions
  - else
    * add the successor list to the corresponding virtual terminal node

Figure 4.11: The control of parser composition by cascading.

all sub-parsers in the same level finish their processing at every node in the LMG, the sub-parsers at one level higher are activated and start to parse the same LMG. This process continues until the highest level is reached. This is a flexible way to deal with extra-grammatical queries even without any sentence-level grammar rules. Since the parsing process starts from level 0 to the highest level, level by level, it is called *cascading*.

- **An Example**

We use an example to illustrate the parser composition by cascading. The partitioned grammars and their corresponding parsing tables in Section 3.5 are used. These sub-grammars are assigned from level 0 to level 1. The OUTPUT of G_0, G_1 and G_2 are S, NP and PP respectively. The input

sentence is "n v det n prep det n". Figure 4.12 shows how the LMG changes by sub-parsers in level 0. First, the input sentence is converted to an LMG, and the terminal nodes are pushed onto the stack θ. Stack θ contains seven elements from top to down in this sequence "n det prep n det v n". Then the sub-parsers of G_1 and G_2 start to parse at n with index 7 in LMG as seen in the first diagram in Figure 4.12. Since sub-parser of G_1 can successfully parse, its output virtual terminal $vtNP$ is created on the LMG, as seen in the first diagram in Figure 4.12. It is then pushed onto the stack θ. Next, the sub-parsers of G_1 and G_2 start to parse at $vtNP$, which is on the top of θ. Since no sub-parsers can successfully parse at $vtNP$, no virtual terminal can be created. $vtNP$ is popped out from stack θ and *det* with index 6 becomes the top of θ. Then they start to parse at *det*. We can trace the sub-parsers how to add virtual terminal on LMG from first to fifth diagrams in Figure 4.12, and sixth diagram is the final stage of LMG after all operations in level 0 . In the following, the control go to level 1. This process continue until master parser is activated at every node in LMG.

4.3.2 Parser Composition with Predictive Pruning

The previous cascading composition algorithm attempts to invoke sub-parsers at every position in the input lattice. In order to avoid excessive invocation, we implemented an alternative parsing algorithm with predictive pruning. Instead of using indices of lattice nodes to determine parsing order, we use a stack σ to store the initial sorted lattice input, and dynamically add newly found virtual terminals to the stack σ in such a way that the topological

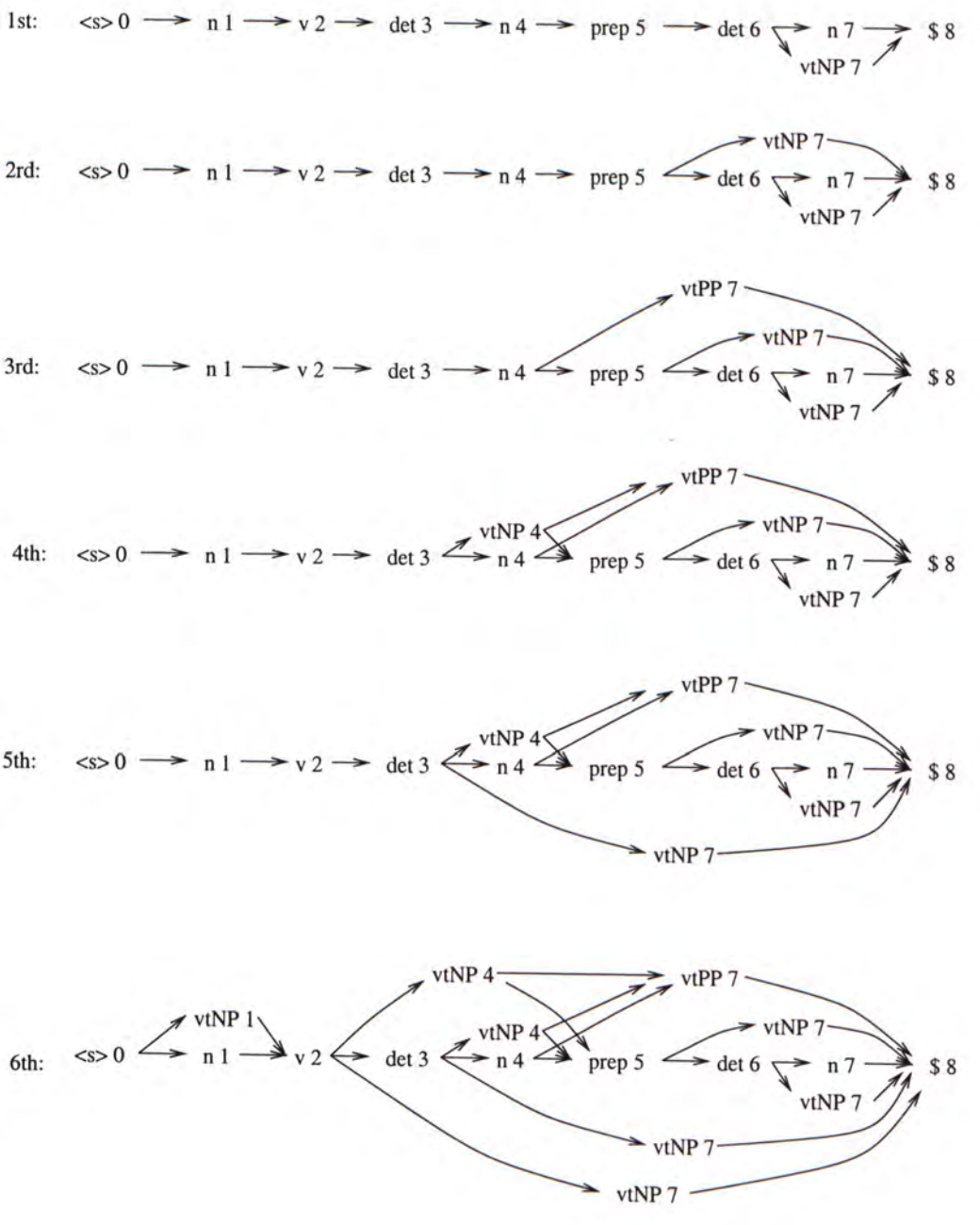

Figure 4.12: Changes in LMG due to sub-parsers in level 0 through parser composition by cascading.

order is always preserved for the changing LMG. Notice that this stack is not GSS. This can avoid repetitive topological sorting.

The detail of this parsing algorithm was presented in [35]. We briefly describes some features of this algorithm here. This algorithm replaces PARSE() and PARSEWORD() in Tomita's algorithm. It mainly contains two functions $psr_m(dt, \sigma)$ and PREDICTIVE-SUB-PARSING(dt, σ), where dt is a (virtual) terminal in a lattice, σ is a stack as mentioned previously and m=0, 1, ..., K.

psr_m is the parser for sub-grammar G_m. psr_0 is the master parser, psr_m for m=1, 2, ..., K are the other sub-parsers. Each psr_m reads the next input from the top of σ and returns a list of virtual terminals if parsed successfully. Otherwise, it returns nothing. psr_m calls PREDICTIVE-SUB-PARSING to add virtual terminals to the stack σ and lattice, and these virtual terminals will be processed. However, our algorithm uses ACTOR(), REDUCER() and SHIFTER() in original Tomita's algorithm instead of Farshi's version [19] of GLR algorithm in psr_m. Therefore, our parser could not deal with grammar having ϵ and cycles.

In PREDICTIVE-SUB-PARSING, $PVT(U_i, w)$ acts as a filter to constrain sub-parser activation. It returns virtual terminal(s), which contains w as their left corner node and can be produced *shift* or *reduce* with U_i. w is a word label of dt, i is the index of dt's left-neighbor and U_i is a layer field of GSS. The relationship between virtual terminal vt and its left corner nodes w can be represented as: $vt \stackrel{+}{\Rightarrow} w$, where $\stackrel{+}{\Rightarrow}$ means "derives in one or more steps".

When a caller sub-parser reads a node in the LMG, the sub-parser that gets activated must have the node as its left corner and must be able to return

Virtual terminal (vt)	Left corner nodes (w)
vtNP	n, det
vtPP	prep

Table 4.1: The left corner node sets of virtual terminals.

a virtual terminal *predicted* by the caller sub-parser in order to successfully proceed. Notice that the left corner node can be either a terminal or a virtual terminal, and the predictive constraints can be loosened when necessary. For our implementation, each virtual terminal's left corner node is precomputed before parsing. All the other sub-parsers in the caller's INPUT set that do not satisfy these predictive conditions are *pruned.* The entire parsing progresses, with the master GLR sub-parser starting at the leftmost lattice node, and it ends when it reaches the final node of the LMG. Since the activated sub-parsers must satisfy the caller sub-parser's predictive constraint and the ones that do not satisfy the constraint are pruned, it is called *predictive pruning.*

- **An Example**

We use the same input query in the sub-section 4.3.1 to illustrate how to compose parsers with predictive pruning. Table 4.1 shows the left corner node sets of virtual terminals of partitioned grammar in Section 3.5. Similar to parser composition by cascading, the input sentence is first converted to an LMG. Then the parser topologically sort the lattice by push all the terminal nodes[2] onto stack σ in the reverse order, so that the resultant stack σ has 8 elements with n on the top and '$' at the bottom of σ.

[2]The LMG only contains terminals before parsing.

At the beginning, master sub-parser (with G_0 as its grammar) is activated to parse the LMG at symbol n, which is the top of stack σ. Sub-parser of G_1 satisfies master-parser's predictive constraints, since ACTION[initial state 0, $vtNP$]$\neq$error from parsing table, and $vtNP \stackrel{+}{\Rightarrow} n$ according to Table 4.1. Therefore, sub-parser of G_1 is activated at symbol n. Since it can successfully parse, it returns a virtual terminal with label $vtNP$ which will be added on the LMG as shown in the first diagram in Figure 4.13, and also pushes $vtNP$ onto stack σ top. Then the sequence of stack σ from top to down becomes: $vtNP$, v, ..., '\$'.

Master sub-parser starts at $vtNP$ now. It shifts $vtNP$ and then v onto its GSS, the stage of GSS is similar to the third diagram in Figure 4.7. Following that reads *det* as the next input. At this stage, sub-parser of G_1 satisfies the predictive condition of master parser, since ACTION[current state 5, $vtNP$]$\neq$error and $vtNP \stackrel{+}{\Rightarrow} det$. Then sub-parser of G_1 is invoked at *det*.

Sub-parser of G_1 starts to parse at *det*. It shifts *det* and n onto its GSS. Then it activates sub-parser of G_2 at *prep* and it can successfully parse the phrase "prep det n". The LMG becomes the third diagram in Figure 4.13. Return to the sub-parser of G_1, it reads $vtPP$ which is on the top of stack σ. Then it can successfully parse and end at '\$'. The resultant LMG is seen as the fourth diagram in Figure 4.13.

Finally, the control return to the master sub-parser and it is going to read $vtNP$s with index 4 or index 7. Two $vtNP$s is shifted onto its GSS one by one and parsing proceeds until '\$' is reached.

When we compare the example of parser composition by cascading in

Section 4.3.1 and this example, we can observe cascading produces more virtual terminals than predictive pruning. The LMG in the sixth diagram of Figure 4.12 contains eight virtual terminals, while the LMG in the fourth diagram of Figure 4.13 contains five virtual terminals. Both of these two algorithms create the same parse forest as shown in Figure 4.10, if the master parser is called.[3] In this example, we see some of virtual terminals created by cascading that are not attached in the resultant parse forest.

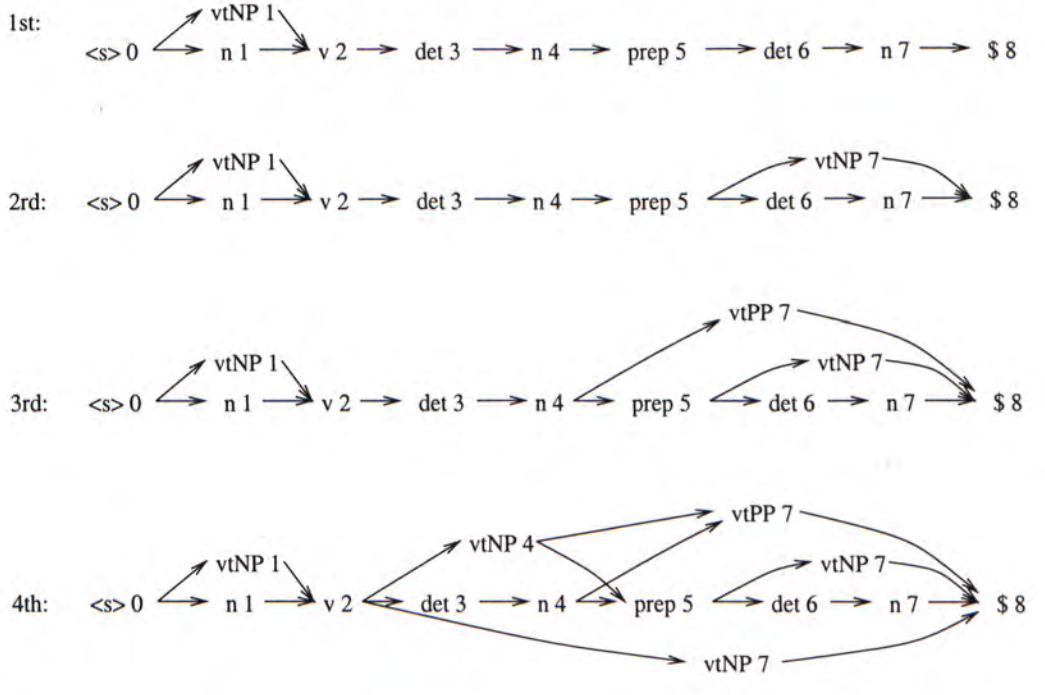

Figure 4.13: Changes in LMG by parser composition with predictive pruning.

[3]If we use one GLR parser for overall grammar to parse this sentence, the same parse forest without virtual terminals can be generated.

4.3.3 Comparison of Parser Composition by Cascading and Parser Composition with Predictive Pruning

After we described these two algorithms in the previous two sections, we try to compare them in the following aspects:

1. Sub-parsers for cascading can activate at any virtual terminal or terminal node in LMG; Sub-parsers (not including master parser) for predictive pruning can only activate at the node that satisfies its predictive constraint.

2. Cascading can proceed without the master parser in its framework. However, predictive pruning requires a master parser, since other sub-parsers are driven by master parser.

3. Cascading does not terminate the parsing process until the highest level sub-parser(s) (or master parser) is parsed. Predictive pruning terminates the parsing process when it detects a grammatical error in the input sentence.

4. Cascading produces more virtual terminals node on LMG than predictive pruning in general, even though some of them do not contribute to final parse forest.

4.4 Chapter Summary

This chapter introduces the LMG, which serves as an interface for the sub-parser operations. Then we described GLR lattice parsing algorithm which

is used by each sub-parser. We also presented two parser composition algorithm to integrate all sub-parsers and discussed the difference between them. We give an example sentences that is parsed by these two parser composition algorithms respectively, and showed parser composition with predictive pruning is comparatively more efficient than cascading composition. In the next chapter, we will try to use our parsing framework on English and Chinese queries in ATIS. We will evaluate parsing performance across partitioned and unpartitioned grammar approach, as well as the two different parser composition algorithms.

Chapter 5

Experimental Results and Analysis

5.1 Introduction

We applied our parsing framework for parsing natural language queries in the Air Travel Information Systems (ATIS) [20][1] corpus, which is widely used in evaluating natural language system. We first introduce how the ATIS grammar was developed for our parser based on training sets. We used (i) a single GLR parser with unpartitioned grammar and (ii) multiple sub-parsers with partitioned grammar to parse ATIS queries. With the GLR parsing algorithm, both cases can provide multiple parses. A semantic interpreter is used to select the best path or parse from the LMG or parse forest, and extract semantic concepts from the parse tree(s).

[1]This research program was sponsored by DARPA. Its corpus is available through LDC membership (http://www/ldc.upenn.edu).

Later in this chapter, we report our experimental results based on degree of grammatical coverage, size of parsing table, computational costs and understanding accuracies across grammar partitioning approach and traditional single GLR approach, and two different two composition algorithms for grammar partitioning.

5.2 Experimental Corpus

The ATIS corpus contains queries inquiring air travel information, such as flights leaving from or arriving at locations at various dates and times, availability of good transportation, airline routes, etc. We translated the English ATIS-3 queries in class-A, a subset of ATIS that are processed individually without context, into Cantonese Chinese, as a parallel corpus for our experiments. Some are in a colloquial Cantonese style, the predominant Chinese dialect used in Hong Kong, South China, Macau and many overseas Chinese communities. There are some example queries in Figure 5.1.

For the parallel ATIS-3 corpus, the training set has 1564 queries, the 1993 test set has 448 queries, and the 1994 test set has 444 queries. Each query is labeled with a corresponding SQL query for database access. Table 5.2 shows an example of a SQL query. The average sentence lengths of the various data sets are shown in Table 5.1. The lengths of English sentences are measured in words, while the lengths of the Chinese sentences are measured in the characters.

English:	*Show me the most expensive one way flight from detroit to westchester country*
Chinese:	話俾我知由底特律飛去西赤斯城最貴既單程航機
English:	*Show me the flights arriving on baltimore on june fourteenth*
Chinese:	話俾我知係六月十四號飛到巴的摩爾既航機
English:	*Show me all united airlines first class flights*
Chinese:	話我知所有聯合航空既頭等航班
English:	*How many first class flights does united have today*
Chinese:	今日有幾多班聯合航空既頭等航機起飛
English:	*Show me all the northwest flights from new york to milwaukee that leave at seven twenty a m*
Chinese:	話俾我知所有係上晝七點廿分由紐約飛去密耳瓦基既航機

Figure 5.1: Examples of English sentences in the ATIS-3 training corpus, together with their Cantonese translations.

ATIS-3 Class A queries	**Training set**	**Test set 93**	**Test set 94**
English Corpus (in words)	11.2	10.3	11.5
Chinese Corpus (in characters)	19.3	16.5	20.1

Table 5.1: Average length of the ATIS-3 queries for the English and Chinese corpora.

Query: show me the flights arriving on baltimore on june fourteenth
SQL:

```
SELECT DISTINCT ""'||v0.flight_id||""', ""'||v1.airport_code||""',
""'||v2.city_code||""', ""'||v2.city_name||""', ""'||v0.flight_days||""',
v0.departure_time, v0.arrival_time, v0.time_elapsed FROM flight v0, air-
port_service v1, city v2
WHERE (((v0.to_airport = ANY v1.airport_code)
AND ((v1.city_code = ANY v2.city_code)
AND (v2.city_name = 'BALTIMORE'))) AND (((v0.flight_days IN (
SELECT v3.days_code FROM days v3
WHERE (v3.day_name IN (
SELECT v4.day_name FROM date_day v4
WHERE (((v4.year = 1993)
AND (v4.month_number = 6)) AND (v4.day_number = 13))))))
AND (((v0.departure_time > v0.arrival_time)
AND ((v0.time_elapsed >= 60) OR (v0.arrival_time < 41)))
AND ( 1 = 1))) OR ((v0.flight_days IN (
SELECT v5.days_code FROM days v5
WHERE (v5.day_name IN (
SELECT v6.day_name FROM date_day v6
WHERE (((v6.year = 1993) AND (v6.month_number = 6))
AND (v6.day_number = 14)))))) AND NOT (((v0.departure_time >
v0.arrival_time)
AND ((v0.time_elapsed >= 60) OR (v0.arrival_time < 41)))
AND ( 1 = 1)))));
```

Figure 5.2: An example of a SQL query for data base access, together with its English query.

5.3 ATIS Grammar Development

The grammar is a set of context-free rules. Our parser requires that the rules are acyclic and have no ϵ-productions. The grammar contains both semantic and syntactic structures. The low level grammar rules are mainly semantic concepts typical of the ATIS corpus, such as CITY-NAME, CLASS-TYPE, MONTH-NUMBER, etc. The high level grammar rules are sentences phrases, such as a time phrase, flight preposition phrase, etc.

The low level grammar rules are obtained by a semi-automatic grammar induction algorithm [25]. This algorithm forms non-terminals bottom-up by means of statistically clustering of un-annotated training corpus, and the output non-terminals are post-processed by hand, e.g. merging and labeling non-terminals as semantic concepts. The set of semantic concepts are characteristic of the domain, and hence common between the English and Chinese grammars.

By applying the parser composition by cascading, sub-parsers with low level indices *IDs* parsed each training sentence to form a lattice. Then SENTENCE-level grammar rules are obtained from the "best" path (shortest-path) through the lattice. The grammar so derived maximizes the coverage of our training set. Thus we formed grammars for the English and Chinese corpora, with unpartitioned and partitioned grammar sizes are listed in Table 5.3. Since the grammars are semi-automatically derived from parallel corpora, the grammar statistics are comparable across languages. Table 5.2 displays a simplified unpartitioned English ATIS-3 grammar. A subset of English and Chinese ATIS-3 grammar is included in Appendix A.

G_S:

* S → QUANT DEPARTURE [TIME_NP]| ...
* DEPARTURE → leaving CITY_NAME [TIME_NP]| ...
* TIME_NP → [on] DAY_NAME| ...
* FLIGHT → flights|flight| ...
* QUANT → all|an|any| ...
* CITY_NAME → phoenix|new york|seattle| ...
* DAY_NAME → monday|tuesday|wednesday| ...

Table 5.2: A subset of English ATIS grammar.

The main difference between English and Chinese queries in the ATIS domain is Chinese needs one or more characters to represent a English word. For example, three Chinese characters "西 雅 圖" to represent "Seattle". Besides, there is no singular or plural concept in Chinese noun. For example, both "flights" and "flight" are translated to "航 班" in Chinese. Therefore, English requires more low-level rules than Chinese. In addition, there is less strict structure in Chinese queries, while compare with English queries. For example, both Chinese sentences "由 他 科 馬 飛 去 蒙 特 利 爾 既 單 程 收 費 有 幾 多 種" and "有 幾 多 種 由 他 科 馬 飛 去 蒙 特 利 爾 既 單 程 收 費" are commonly used to represent "how many fares are there one way from Tacoma to montreal". In English, the question subject (such as "how many fares") usually appears at the beginning of the sentences. This makes Chinese requires more SENTENCE-level rules than English.

Grammar Statistics	No Partitioning		Partitioned Grammar	
	English	Chinese	English	Chinese
Number of rules	1650	1538	1818	1637
Number of SENTENCE-level rules	337	508	337	508
Number of terminals	602	515	602	515
Number of non-terminals	97	85	97	85
Number of virtual terminals	N/A	N/A	65	63
Total number of states in parsing table	72,869	29,734	3,350	3,894

Table 5.3: Grammar statistics based on the original unpartitioned grammar and partitioned grammar.

5.4 Grammar Partitioning and Parser Composition on ATIS Domain

5.4.1 ATIS Grammar Partitioning

Based on our grammar partitioning scheme, we manually partitioned our rule sets into sub-grammars by choosing non-terminals, such as STATE-NAME, CITY-NAMES, AIRPORT-CODE, etc., with corresponding virtual terminals prefixed with *vt*. Most of these non-terminals selected are semantic concepts, so that they can be presented in the lattice. Each sub-grammar is assigned with *ID*, according to level ordering criteria in Section 3.4. The calling graph of our partitioned grammars is a directed acyclic graph (DAG).

The sizes of the unpartitioned and partitioned grammars are listed in Table 5.3. We partitioned the English and Chinese ATIS grammar into 65

and 63 sub-grammars respectively. Both English and Chinese sub-grammars are assigned from level 0 to level 8. A sub-grammar is assigned to the 0th level (i.e. ID=0) if all nonterminals on the LHS of its production rules are preterminals. There is only one sub-grammar with level 8 for each English and Chinese grammar sets, which is the master grammar with SENTENCE-level rules.

We observe from Table 5.3 (first row) that the partitioned English grammar has 1818 rules, and the unpartitioned English grammar has 1650 rules. Hence the partitioned grammar seems larger than the unpartitioned. This is because some rules have been duplicated across multiple sub-grammars. The duplicated non-terminals do not constitute the output of any sub-grammars. For example, production rule DIGIT $\rightarrow$ *one* may appear in sub-grammar of COST, TIME, etc, but DIGIT is not the OUTPUT of any sub-grammar.

Table 5.4 shows an example of partitioned grammars for the subset of English ATIS-3 grammar as shown in Table 5.2. The sub-grammar G_S in level 3 acts as a master grammar. Figure 5.3 shows an an example of calling graph of the sub-grammars after partition, which is a DAG.

5.4.2 Parser Composition on ATIS

After we partitioned the ATIS grammar, we use two parser composition algorithms to integrate sub-parsers – parser composition by cascading and parser composition with predictive pruning. Given a ATIS query such as "all flights leaving phoenix on wednesday". The partitioned grammars in Table 5.4 are used. The output LMG from cascading and predictive pruning are shown in

G_S: (level 3)
INPUT = {vtDEPARTURE, vtQUANT, vtTIME_NP, ... }
OUTPUT = {S}
* S → vtQUANT vtDEPARTURE [vtTIME_NP]| ...

$G_{DEPARTURE}$: (level 2)
INPUT = {leaving, vtCITY_NAME, vtTIME_NP, ... }
OUTPUT = {DEPARTURE}
* DEPARTURE → leaving vtCITY_NAME [vtTIME_NP]| ...

G_{TIME_NP}: (level 1)
INPUT = {vtDAY_NAME, on, ... }
OUTPUT = {TIME_NP}
* TIME_NP → [on] vtDAY_NAME|[on] vtDATE| ...

G_{FLIGHT}: (level 1)
INPUT = {flights, ... }
OUTPUT = {FLIGHT}
* FLIGHT → flights|flight| ...

G_{QUANT}: (level 0)
INPUT = {all, ... }
OUTPUT = {QUANT}
* QUANT → all|an|any| ...

G_{CITY_NAME}: (level 0)
INPUT = {phoenix, ... }
OUTPUT = {CITY_NAME}
* CITY_NAME → phoenix|new york|seattle| ...

G_{DAY_NAME}: (level 0)
INPUT = {wednesday, ... }
OUTPUT = {DAY_NAME}
* DAY_NAME → monday|tuesday|wednesday| ...

Table 5.4: A subset of partitioned English ATIS grammar.

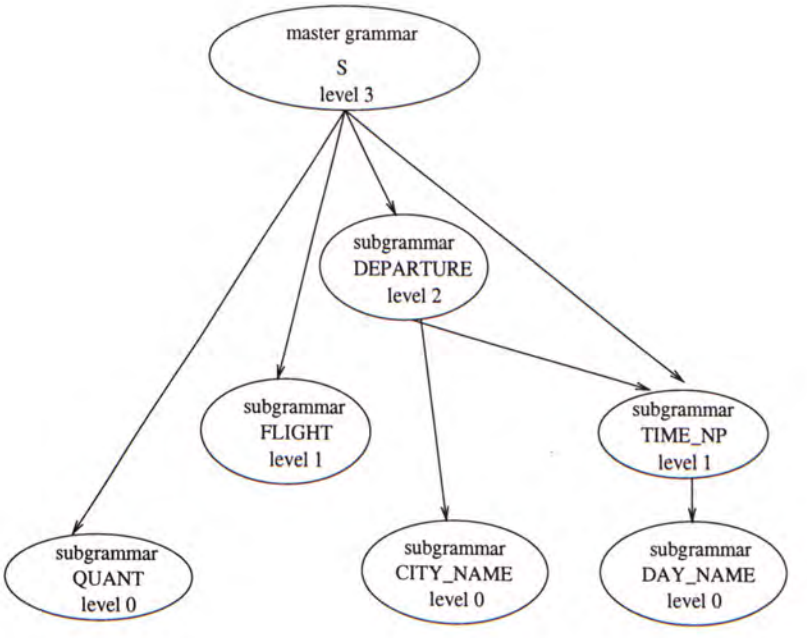

Figure 5.3: The calling graph of the sub-grammars in Table 5.4.

Figure 5.4 and Figure 5.5 respectively. Notice that using predictive pruning can save one virtual terminal *vt*TIME_NP on LMG for the same output parse in Figure 5.7.

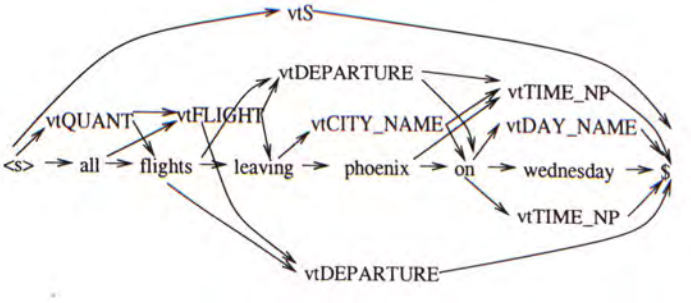

Figure 5.4: The resultant LMG by using parser composition by cascading.

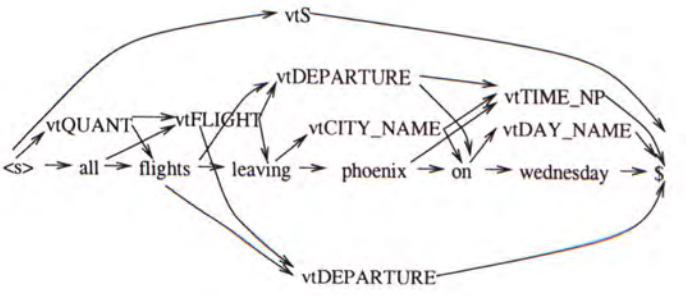

Figure 5.5: The resultant LMG by using parser composition with predictive pruning.

5.5 Ambiguity Handling

A GLR parser is able to handle ambiguity and provide multiple parses. GLR parsers act as a core component in our parsing framework, our parser can also handle ambiguity. Our parsing framework can produce the same parses as a single GLR parser with the same set of grammar rules. In this section, we provide some example queries from that contains multiple parses.

The input query: "what does fare code h mean" can generate two parse trees, which are shown in Figure 5.6. The ambiguity of this sentence is due to two representation of 'h' – booking class and fare basic code. In this case, these two representation are possible.

Another input query: "all flights leaving phoenix on wednesday" can also produce two parse trees, as seen in Figure 5.7. The ambiguity of this sentence is due to generalization of SENTENCE-level rules. In this case, the parse tree that contains 3 leaf nodes is better than the one that contains 4 leaf nodes. It is because the former one can represent "wednesday" as a departure day, the latter one treats "on wednesday" be a prepositional phrase attached to

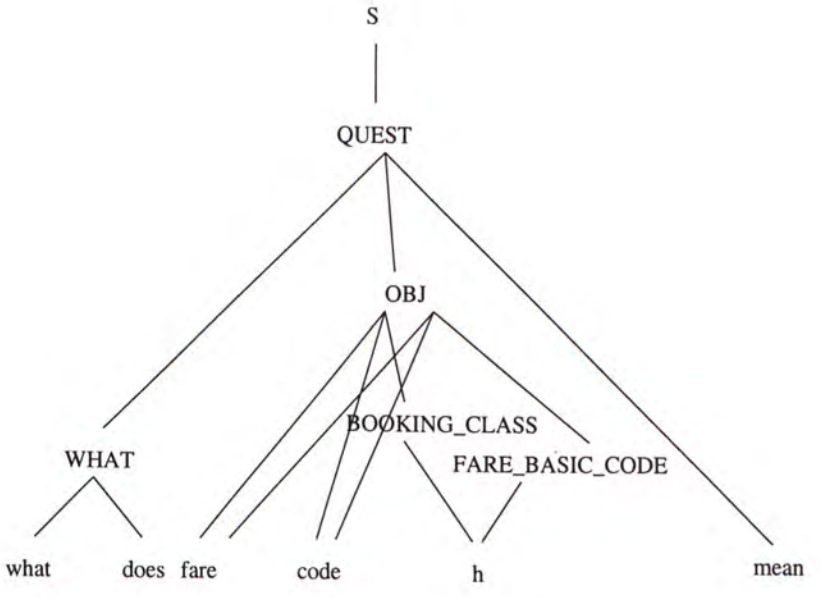

Figure 5.6: Example of a multiple parses sentence.

"all flights leaving phoenix".

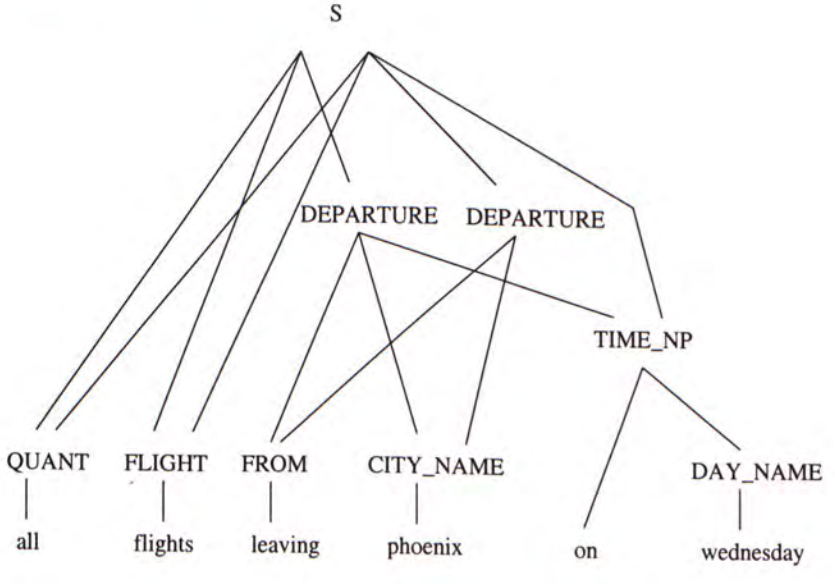

Figure 5.7: Another example of multiple parses sentence.

For this reason, we select a parse from sentence node S if it contains

more than one parses. For grammar partitioning approach, a shortest-path algorithm is used to select the "best" path. The detail is described in the next section. For a single GLR, we select a parse with minimum number of leaf nodes if multiple parses are obtained. Then some non-terminal inner nodes of the parse tree can cover more terminal nodes, this manner is similar to finding the shortest path in grammar partitioning approach.

5.6 Semantic Interpretation

5.6.1 Best Path Selection

With grammar partitioning, the output of our parsing framework is an LMG. There are multiple paths from the sentence START to sentence END to represent the original sentence. We apply the shortest-path algorithm in [10] to find the best path in the LMG for representing the input sentence.

The shortest-path algorithm is to find the path which connects sentence START <s> and sentence END '$', and has minimum total distance. This algorithm starts from the sentence START to successively identify the directed path to each nodes, until the sentence END is reached. Figure 5.9 shows the algorithm for shortest path problem [10].

The transitions in our LMG are equally weighted, i.e. the distance between any two direct connected (virtual) terminal nodes are the same (distance=1). Figure 5.8 shows a subset of nodes in an LMG. There are two paths from terminal *from* to *to* – one path passed through *vt*CITY_NAME and another path passed through *seattle*.

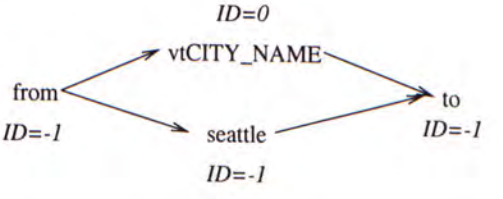

Figure 5.8: A subset of nodes in a LMG.

The *ID* of sub-grammars are assigned during partitioning grammar. The virtual terminal nodes in an LMG are assigned with the same *ID* with their

sub-grammars. The terminal nodes in an LMG are assigned with $ID = -1$. The IDs of virtual terminals and terminals in LMG are indicated in Figure 5.8.

However, in most cases, we prefer finding a path passed through a node with relative high *ID*. For the example in Figure 5.8, we prefer the path passed through *vt*CITY_NAME, since level index of *vt*CITY_NAME (*ID*=0) is higher than *seattle* (*ID*=−1). Then we scale the distance by multiplying it by a level cost. The higher the level number of the target node, the lower the level cost of the transition. Equation (5.1) is used to compute the level cost.

$$c = t - i \tag{5.1}$$

where

c denotes level cost of a transition,

t denotes total number of level,

i denotes the target node level index.

Objective of nth iteration: Find nth nearest node to origin. (To be repeated for n=1, 2, ..., until nth nearest node is the destination.)

Input for nth iteration: (n-1) nearest nodes to origin (solved for at previous iterations), including their shortest path and distance from the origin. (These nodes, plus the origin, will be called *solved nodes*; the others are *unsolved nodes*.)

Candidates for nth nearest node: Each solved node that is directly connected by a link to one or more unsolved nodes provides *one* candidate – the unsolved node with the *shortest* connecting link. (Ties provide additional candidates.)

Calculation of nth nearest node: For each such solved node and its candidate, add the distance between them and the distance of the shortest path from the origin to this solved node. The candidate with the smallest such total distance is the nth nearest node (ties provide additional solved nodes), and its shortest path is the one generating this distance.

Figure 5.9: Algorithm for shortest path problem [10].

5.6.2 Semantic Frame Generation

A semantic frame contains a list of key-value pair(s) to represent the meaning of the sentence. The keys are designed according to the schema of SQL of ATIS queries, such as CITY-NAME, FLIGHT-NUMBER, CLASS-TYPE. After the best path is selected from the LMG, only virtual terminals are considered and terminals are ignored. Because virtual terminals carry semantic content. The virtual terminal nodes in an LMG actually corresponds to a sub-tree (or multiple sub-trees) in the paired shared parse forest. But the sub-tree is not displayed in the LMG. The semantic interpretor walks through the parse

trees which attached to the virtual terminals in the best path. If the root node or inner node of the parse tree matches with the keys, it extracts the terminal(s) under the key. The extracted terminal(s) becomes the value of the key-value pair. There is an operand represents the relationship between the key and value, such as '=', '<', '$\sim$', '$\leq$', etc.

Figure 5.10 shows an example parse tree. The keys are DEPARTURE, TIME, CITY_NAME, then the semantic frame is shown in Table 5.11.

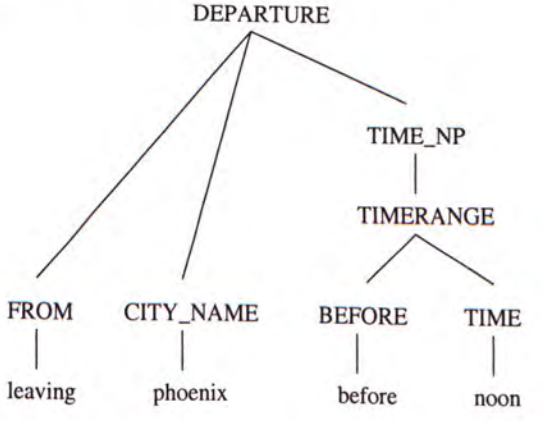

Figure 5.10: The parse tree is attached to a virtual terminal vtDEPARTURE.

DEPARTURE_TIME<noon
DEPARTURE_CITY_NAME=phoenix

Figure 5.11: Example of a semantic frame.

5.6.3 Post-Processing

Post-processing is included to refine the generated semantic frame, which is designed according to the training sets. There are two post-processing

heuristics: (i) if the frame contains a TIME key, but we cannot determine whether it is DEPARTURE_TIME or ARRIVAL_TIME, we will default to DEPARTURE_TIME. (ii) if the frame contains a COST key, we will classify it as ONE_DIRECTION_COST or ROUND_TRIP_COST according to the other keyword appeared in the sentence, such as "round trip", "nonstop", etc.

5.7 Experiments

Our experiments compare the two parsing approaches: (i) single GLR for the overall unpartitioned grammar, (ii) parser composition by cascading and parser composition with predictive pruning. Comparison is based on grammar coverage, parsing table sizes, computational costs and understanding accuracies.

For the tables in this section, CAS abbreviates parser composition by cascading, PP abbreviates for parser composition with predictive pruning and GLR abbreviates for single GLR parser approach.

5.7.1 Grammar Coverage

Experiments are conducted with ATIS-3 queries. Grammar coverage of English and Chinese ATIS are shown in Table 5.5 and Table 5.6 respectively. *Full parse* means there is a parse forest covering the whole query. *Partial parse* means there is at least one parse chunk covering part of query. *No parse* means there is no parse at all for the query.

Full parse coverage for English sentences vary between 60% to 62%, while

that for Chinese sentences are lower at 44% to 57%. Since our SENTENCE-level grammar rules are derived from shortest paths through the training lattices, they tend to be rather specific in structure, and may contribute to the relatively low full parse coverage. We observe from Table 5.3 (second row) that Chinese grammar has 508 SENTENCE-level rules, and English grammar has 337 SENTENCE-level rules. English grammar requires fewer SENTENCE-level rules to obtain comparable grammar coverage in the training set. This implies English grammar is more general and can obtain higher grammar coverage in the test sets as shown in Table 5.5 and 5.6.

Since we use the same grammar across different parsing methods, percentage of full parse queries are the same in English and Chinese respectively. For single GLR parser approach, it cannot create any partial parse. For the parser composition by cascading, sub-parsers can be activated and allows a parse to end at any position of the input query. A sub-parser creates virtual terminal node on an LMG if it can successfully parse. For parser composition with predictive pruning, sub-parsers only can be activated if they satisfy the predictive pruning conditions. The parsing process terminates when the master parser detects an error from the parsing table. Therefore, cascading obtains the highest percentage of partial parse and zero percentage of no parse queries in test sets 93 and 94.

5.7.2 Size of Parsing Table

We use the same LR(1) parsing table generator to construct an LR(1) table for the unpartitioned English and Chinese grammars. We also construct 65

Based on the English ATIS-3 Corpus	Training set			Test set 93			Test set 94		
	GLR	CAS	PP	GLR	CAS	PP	GLR	CAS	PP
Full parse (%)	99.4	99.4	99.4	60.9	60.9	60.9	62.4	62.4	62.4
Partial parse (%)	0	0.6	0.5	0	39.1	34.2	0	37.6	35.1
No parse (%)	0.6	0	0.1	39.1	0	4.9	37.6	0	2.5

Table 5.5: Grammar coverage for the English ATIS-3 corpora.

Based on the Chinese ATIS-3 Corpus	Training set			Test set 93			Test set 94		
	GLR	CAS	PP	GLR	CAS	PP	GLR	CAS	PP
Full parse (%)	98.7	98.7	98.7	44.6	44.6	44.6	57.4	57.4	57.4
Partial parse (%)	0	1.3	1.0	0	55.4	51.3	0	42.6	37.4
No parse (%)	1.3	0	0.3	55.4	0	4.0	42.6	0	5.2

Table 5.6: Grammar coverage for the Chinese ATIS-3 corpora.

tables for the partitioned English grammar and 63 tables for the partitioned Chinese grammar. The total number of states (rows) in the parsing table for partitioned grammars is the sum of the number of states in all sub-parsers. For unpartitioned grammar, there is a single parsing table. From Table 5.3 (last row), the unpartitioned English grammar has 72,869 states and the unpartitioned Chinese grammar has 29,734 states in their parsing tables. In comparison, the partitioned English grammar has 3,350 states and the partitioned Chinese grammar has 3,894 states in total. Grammar partitioning greatly economizes on parsing table sizes. This decreases the computation required to generate the parsing tables, space or memory to store the tables,

and time to access the tables during parsing. This result shows that the simple LR parsing technique becomes impractical for large grammars for natural language processing. We can also see that with a comparable number of rules, the unpartitioned English ATIS grammar obtained a higher parse coverage than the unpartitioned Chinese grammar, for both test sets. This may suggest that the English rules are more general than the Chinese rules. We also noticed that the number of parsing table states for English is twice that for Chinese. Hence the English parsing table is more expensive to store and access. From this we can see that the parsing table size is dependent not only on the number of grammar rules (grammar size), but also the structure of the grammar rules.

5.7.3 Computational Costs

We compared the computational costs between the two parser composition algorithms, measured in terms of the total parsing time, the number of states visited, and the number of rules reduced. Results are tabulated in Table 5.7 and Table 5.8 for English and Chinese ATIS respectively.

The total parsing time does not include the load time for data, such as parsing table, it is simply the amount of CPU time used by the parsing process. All experiments in this section are conducted on a Sun Solaris Ultra 5 machine in order to minimize time variation due to different system architectures.

From Table 5.7 and Table 5.8, we observed that the single GLR parser is the most economical parsing strategy, when compared with the other two

Based on the English ATIS-3 Corpus	Training set			Test set 93			Test set 94		
	GLR	CAS	PP	GLR	CAS	PP	GLR	CAS	PP
Total parsing time (in seconds, Sun Ultra 5)	42.8	82.2	52.4 *(36.3%)*	8.2	19.2	10.9 *(43.2%)*	9.7	21.7	12.8 *(41.0%)*
Average no. of states visited included failed sub-parsers	60.0	134.2	95.0 *(29.2%)*	37.5	107.8	60.6 *(43.8%)*	47.4	122.9	75.7 *(38.4%)*
Average no. of rules reduced included failed sub-parsers	34.5	67.5	46.6 *(31.0%)*	21.9	52.1	28.4 *(45.5%)*	27.2	59.4	36.5 *(39.6%)*
Average no. of states visited, successful sub-parsers only	59.8	109.7	84.3 *(23.2%)*	32.6	85.1	51.9 *(39.0%)*	39.4	97.0	65.5 *(32.5%)*
Average no. of rules reduced, successful sub-parsers only	34.4	67.5	46.5 *(31.1%)*	19.6	52.1	28.4 *(45.5%)*	23.3	59.3	36.5 *(38.4%)*

Table 5.7: Computational costs for the English ATIS-3 corpora. Italicized percentages in parentheses are savings of PP relative to CAS.

parsing methods. This is true for both Chinese and English ATIS. Our observation is based on the following measurements: the total parsing times, the number of states visited and the number of rules reduced. This is because the single GLR only handles a string as input, but the composed subparsers need to handle a lattice as input. Parsing a lattice involves searching for the left-neighbor of the input symbol in the GSS, and hence is more computa-

Based on the Chinese ATIS-3 Corpus	Training set			Test set 93			Test set 94		
	GLR	CAS	PP	GLR	CAS	PP	GLR	CAS	PP
Total parsing time (in seconds, Sun Ultra 5)	30.2	85.2	51.0 *(40.1%)*	5.7	17.9	11.5 *(35.8%)*	6.7	25.3	12.4 *(51.0%)*
Average no. of states visited included failed sub-parsers	54.9	138.8	81.4 *(41.4%)*	34.4	99.4	54.8 *(44.9%)*	42.4	145.1	65.9 *(54.6%)*
Average no. of rules reduced included failed sub-parsers	22.1	43.7	25.1 *(42.6%)*	13.6	29.9	16.0 *(46.5%)*	16.7	44.8	19.6 *(56.3%)*
Average no. of states visited, successful sub-parsers only	54.5	96.9	67.5 *(30.3%)*	21.9	65.0	42.0 *(35.4%)*	33.6	98.4	51.9 *(47.3%)*
Average no. of rules reduced, successful sub-parsers only	22.0	43.5	25.0 *(42.6%)*	9.0	29.7	16.0 *(46.1%)*	13.7	44.6	19.6 *(56.1%)*

Table 5.8: Computational costs for the Chinese ATIS-3 corpora. Italicized percentages in parentheses are savings of PP relative to CAS.

tionally expensive, as reflected by our measurements. Besides, partitioned sub-parsers insert a robust END to active state nodes in the GSS during parsing. This robustness feature would also increase the parsing time.

However, we should note that the single GLR parser has the largest parsing table. This will require a much longer loading time. Furthermore, runtime parsing will also slow down if the parser needs to request a lot of mem-

ory from the machine. Therefore from this perspective the single GLR parser may be impractical.

As we migrated from cascading to predictive pruning, we observed consistent improvements in all aspects of the parsing computation. For predictive pruning, the master GLR sub-parser (SENTENCE-level sub-parser) starts at the leftmost lattice nodes. For cascading, we also restricted the master sub-parser (SENTENCE-level sub-parser) starts at leftmost lattice nodes. We try to treat this two algorithms in the same manner to provide a fair comparison. The total parsing times for the test sets are shortened by 35.8% to 51.0%, and the trend maintains for the subset of the sentences with full parses, i.e., the parsing times are shortened by 19.0% to 38.9%. The results are tabulated in Table 5.9 and Table 5.10 for English and Chinese ATIS. The number of states visited is reduced by 25.4% to 41.8% when failed sub-parsers are counted, and by 26.6% to 46.2% when only successful sub-parsers are counted. Finally, the number of rules reduced are lower by 19.8% to 36.3% when failed sub-parsers are counted, and by 26.5% to 46.2% when only successful sub-parsers are counted. Notice that for training sets, their grammar coverage are identical, and therefore, their parsing times are reflected in Table 5.7 and Table 5.8.

Chinese also seems to have a greater gain than English. This is because the Chinese word consists of one or more characters, and as we parse character by character in Chinese, we often encounter characters that can be derived form multiple sub-grammars. For example, 三 may be found in city name 三藩市 (San Francisco), time 三點 (three o'clock) as well as airline

name 三角州航空 (Delta). Composition using predictive pruning provides stronger word constraints for the selection of an appropriate sub-grammar.

Based on the English ATIS-3 Corpus	**Test set 93 (273 queries)**			**Test set 94 (277 queries)**		
	GLR	**CAS**	**PP**	**GLR**	**CAS**	**PP**
Total parsing time (in seconds, Sun Ultra 5)	7.2	11.9	8.9 *(25.2%)*	8.4	14.0	10.1 *(27.9%)*
Average no. of states visited included failed sub-parsers	53.5	113.0	84.1 *(25.6%)*	63.2	131.7	98.2 *(25.4%)*
Average no. of rules reduced included failed sub-parsers	32.2	56.5	41.2 *(27.1%)*	37.3	67.3	49.4 *(26.6%)*
Average no. of states visited, successful sub-parsers only	53.5	92.7	74.3 *(19.8%)*	63.2	109.1	87.5 *(19.8%)*
Average no. of rules reduced, successful sub-parsers only	32.2	56.4	41.1 *(27.1%)*	37.3	67.2	49.4 *(26.5%)*

Table 5.9: Computational costs for the subsets of sentences in test set of English ATIS-3 corpora with full parses. Italicized percentages in parentheses are savings of PP relative to CAS.

Based on the Chinese ATIS-3 Corpus	Test set 93 (200 queries)			Test set 94 (255 queries)		
	GLR	CAS	PP	GLR	CAS	PP
Total parsing time (in seconds, Sun Ultra 5)	3.5	8.4	6.8 *(19.0%)*	5.2	14.9	9.1 *(38.9%)*
Average no. of states visited included failed sub-parsers	49.0	108.2	73.7 *(31.9%)*	58.6	156.5	87.2 *(41.8%)*
Average no. of rules reduced included failed sub-parsers	20.2	33.8	22.7 *(32.8%)*	23.8	50.7	27.3 *(46.2%)*
Average no. of states visited, successful sub-parsers only	49.0	75.0	60.0 *(20.0%)*	58.6	113.0	72.0 *(36.3%)*
Average no. of rules reduced, successful sub-parsers only	20.2	33.8	22.7 *(32.8%)*	23.8	50.7	27.3 *(46.2%)*

Table 5.10: Computational costs for the subsets of sentences in test set of Chinese ATIS-3 corpora with full parses. Italicized percentages in parentheses are savings of PP relative to CAS.

5.7.4 Accuracy Measures in Natural Language Understanding

Our semantic interpreter extracts semantic information to form a semantic frame. The contents of the frame are compared against with "reference" semantic frame, which is derived from the list of attributes of corresponding of our parse outputs. For the example SQL in Table 5.2, the underlined attributes are the key-value pairs of reference semantic frame. Thus we can evaluate the performance in natural language understanding of our parse outputs. For each query, we measured in term of the error rate for comparing

the understanding accuracy. First of all, matching accuracy is calculated by Equation (5.2). We also accounted for the insertion error by Equation (5.3) if number of keys in our frame is *more than* that in reference frame. Otherwise, the insertion error is zero. Equation (5.4) is the error rate for each query.

$$ma = \frac{mk}{rk} \tag{5.2}$$

$$ie = \frac{ok - rk}{rk} \tag{5.3}$$

$$er = 1 - (ma - ie) \tag{5.4}$$

where

ma denotes matching accuracy,

ie denotes insertion error,

er denotes error rate,

mk is no. of matched keys for our frame and reference frame,

rk is no. of keys in reference frame,

ok is no. of keys in our frame.

Full match refers to queries with exact matches between the generated semantic frame and the reference semantic frame. In the other words, the error rate is zero. *Partial match* refers to the situation when the error rate is between zero and one for the sentence, with the error types being insertion, deletion, and substitution. *No match* refers to the situation when the error rate equals or exceeds 100% for the sentence. There are some example queries extracted from ATIS test set 1993 for full match and partial match. For full match, the example query is: "i would like to book a round trip flight from kansas city to chicago" (sentence 1). Generated semantic frame and the corresponding reference frame are shown in Figure 5.12 and 5.13 respectively. The key-value pairs in the generated frame with symbol $\surd$ can match with the key-value pairs in reference frame. The partial matched example query is: "find american flight from newark to nashville around six thirty p m" (sentence 2). Generated semantic frame and the corresponding reference frame are shown in Figure 5.14 and 5.15 respectively. In this case, our semantic frame is missing one key-value pair.

Results of matching accuracy are shown in Table 5.11 and Table 5.12 for English and Chinese ATIS respectively. We observe that the error rates of English ATIS is lower than for Chinese. This is due to translation errors. For example, the word "nonstop" is translated to 直航 (direct flight) instead of 不停站 (non-stop). The translation may also be missing some key concepts. Besides, the parser may not recognize all representations of a translated word, e.g. San Diego may be translated to 聖地亞哥 or 聖地牙哥.

For the sentences with full parse, there may not could be a full match

Based on the English ATIS-3 Corpus	Training set			Test set 93			Test set 94		
	GLR	CAS	PP	GLR	CAS	PP	GLR	CAS	PP
Error rate in semantic concepts (%)	8.0	7.5	7.8	41.2	9.9	33.8	40.4	11.7	29.1
Full match (% of sentences)	82.4	82.7	82.4	56.3	87.7	60.5	54.5	77.0	59.2
Partial match (% of sentences)	15.0	15.3	15.2	4.0	8.0	9.8	7.7	18.9	20.3
No match (% of sentences)	2.6	2.0	2.4	39.7	4.2	29.7	37.8	4.1	20.5

Table 5.11: Performance in language understanding of the English ATIS-3 corpora.

Based on the Chinese ATIS-3 Corpus	Training set			Test set 93			Test set 94		
	GLR	CAS	PP	GLR	CAS	PP	GLR	CAS	PP
Error rate in semantic concepts (%)	10.3	7.9	8.6	58.6	11.0	25.6	44.9	12.7	28.4
Full match (% of sentences)	77.6	80.8	80.2	37.5	78.6	61.6	49.8	72.7	58.6
Partial match (% of sentences)	18.5	16.9	17.1	6.0	16.7	23.2	7.4	22.3	24.1
No match (% of sentences)	3.9	2.2	2.7	56.5	4.7	15.2	42.8	5.0	17.3

Table 5.12: Performance in language understanding of the Chinese ATIS-3 corpora.

for its semantic interpretation. One reason is due to some implicit information inferred from the query appears on the reference frame. For example, the word "tonight" represents "time>=1800" and "time<=2359" today, "time>=0" and "time<=600" in the following day. However, we could only generated one key-value pair "time=tonight" in our frame.

Our semantic frame: (error rate=0)
DEPARTURE_CITY_NAME=kansas city√
ARRIVAL_CITY_NAME=chicago√
ROUND_TRIP_COST=round trip√

Figure 5.12: The semantic frame generated for the query (sentence 1) "i would like to book a round trip flight from kansas city to chicago." This frame fully matches the reference semantic frame in Figure 5.13, and is counted as a "full match" in Table 5.11.

SQL reference frame:
city_name='KANSAS CITY'
city_name='CHICAGO'
round_trip_cost IS NOT NULL

Figure 5.13: The reference frame for the query (sentence 1) "i would like to book a round trip flight from kansas city to chicago," generated from SQL.

Our semantic frame: (error rate=0.8)
AIRLINE_CODE=american√
DEPARTURE_CITY_NAME=newark√
ARRIVAL_CITY_NAME=nashville√
DEPARTURE_TIME~six thirty p m√

Figure 5.14: The semantic frame generated for the query (sentence 2) "find american flight from newark to nashville around six thirty p m." This frame fully matches the reference semantic frame in Figure 5.15, and is counted as a "partial match" in Table 5.11.

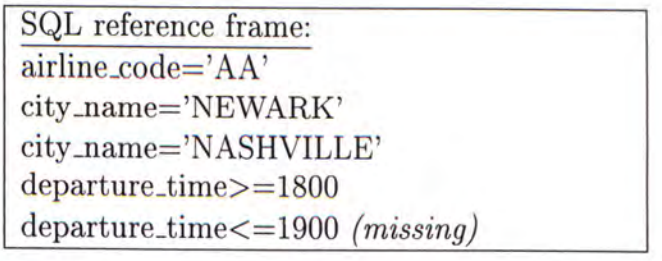

Figure 5.15: The reference frame for the query (sentence 2) "find american flight from newark to nashville around six thirty p m." generated from SQL.

From Table 5.11 and Table 5.12, we found that the single GLR method obtains the highest error rate in matching semantic concepts, since the single GLR parser obtains no parse if the input query is grammatically incorrect. However, partitioned grammar approach may provides partial parse on LMG even the sentence is grammatical wrong. For the training set (first to third column), these three parsing strategies do not obtain the same error rate due to the different partial coverage.

In addition, performance on language understanding suffers a decline as we shifted from cascading to predictive pruning. This is highly correlated with the grammar coverage. Cascading attempts to parse chunks of the input at all lattice nodes, while predictive pruning invokes virtual terminals only if they abide to the left corner predictive constraints.

If we focus on the subset of test sentences with full parse, we expect that they should have comparable language understanding performance. However, we see from Tables 5.13 and 5.14 that the single GLR parser has higher error rates in language understanding. It should be noted that the semantic frame is generated different for the single GLR parser, when compared to the composed subparsers. In the former, we selected the parser tree with the

fewest leaf nodes. For the latter, we ran the shortest-path algorithm through the LMG. This has been described previously.

The performances on language understanding are identical across languages and composition algorithms. This is true except for the single GLR approach.

Except for the single GLR approach, there is a little variation due to different best parse selection methods as discussed in Section 5.6.1. The results are displayed in Table 5.13 and Table 5.14 for English and Chinese respectively.

However, cascading generates more partial parses than predictive pruning, and the single GLR parser cannot generate any partial parse as discussed in Section 5.4.1. If we focus on the subset of test sentences with partial parses, cascading salvages many more to attain full understanding, when compared to predictive pruning. This resulted in a significant difference in the overall performance in language understanding as shown in Table 5.15 and Table 5.16. For example, in the 1993 Chinese test set, there were 200 full parse sentences, of which 171 were fully understood. Cascading produced 248 partial parses of which 181 were fully understood, while predictive pruning method produced 230 partial parses of which only 105 were fully understood. Overall this constitutes a difference of 17.0% (a decline from 78.6% to 61.6% in Table 5.12) on fully understood queries. The other test sets shows similar trends. These results suggests the need for more versatile SENTENCE-level grammar rules, as well as enhanced robustness in parsing should predictive pruning composition be adopted.

Based on the English ATIS-3 Corpus	Test set 93 (273 queries)			Test set 94 (277 queries)		
	GLR	CAS	PP	GLR	CAS	PP
Error rate in semantic concepts (%)	3.5	3.5	3.5	4.5	4.4	4.4
Full match (% of sentences)	92.3	92.3	92.3	87.4	87.7	87.7
Partial match (% of sentences)	6.6	6.6	6.6	12.3	11.9	11.9
No match (% of sentences)	1.1	1.1	1.1	0.4	0.4	0.4

Table 5.13: Language understanding performance for the subsets of sentences in test set of English ATIS-3 corpora with full parses.

Based on the Chinese ATIS-3 Corpus	Test set 93 (200 queries)			Test set 94 (255 queries)		
	GLR	CAS	PP	GLR	CAS	PP
Error rate in semantic concepts (%)	7.3	6.7	6.7	4.1	4.1	4.1
Full match (% of sentences)	84	85.5	85.5	86.7	86.7	86.7
Partial match (% of sentences)	13.5	12.0	12.0	12.9	12.9	12.9
No match (% of sentences)	2.5	2.5	2.5	0.4	0.4	0.4

Table 5.14: Language understanding performance for the subsets of sentences in test set of Chinese ATIS-3 corpora with full parses.

Based on the English ATIS-3 Corpus	Test set 93			Test set 94		
	GLR	CAS	PP	GLR	CAS	PP
Number of queries	0	175	153	0	167	156
Error rate in semantic concepts (%)	N/A	19.7	78.2	N/A	23.9	68.2
Full match (% of sentences)	N/A	80.6	12.4	N/A	59.3	12.8
Partial match (% of sentences)	N/A	10.3	17.0	N/A	30.5	36.5
No match (% of sentences)	N/A	9.1	70.6	N/A	10.2	50.6

Table 5.15: Language understanding performance for the subsets of sentences in test set of English ATIS-3 corpora with partial parses.

Based on the Chinese ATIS-3 Corpus	Test set 93			Test set 94		
	GLR	CAS	PP	GLR	CAS	PP
Number of queries	0	248	230	0	189	166
Error rate in semantic concepts (%)	N/A	14.5	36.2	N/A	24.4	55.9
Full match (% of sentences)	N/A	73.0	45.7	N/A	54.0	23.5
Partial match (% of sentences)	N/A	20.6	34.8	N/A	34.9	44.6
No match (% of sentences)	N/A	6.5	19.6	N/A	11.1	31.9

Table 5.16: Language understanding performance for the subsets of sentences in test set of Chinese ATIS-3 corpora with partial parses.

5.7.5 Summary of Results

In this section, we summarize the experimental results in the different dimensions. We compare the difference between (i) the multiple parsers and single GLR parser approaches; (ii) parser composition by cascading and parser composition with predictive pruning; and (iii) English and Chinese ATIS grammars.

First, we compare the difference between multiple parsers and single GLR parser approaches. Grammar partitioning significantly reduce the size of parsing table, i.e. this save time to construct and space to store the parsing table. We have provided strong evidences that single GLR approach is not practical for large grammars. In addition, composing multiple parsers for parsing can obtain higher understanding accuracies that contributed by partial parses. However, composing parsers requires higher computational costs due to robustness.

Second, we compare the difference between parser composition by cascading and parser composition with predictive pruning. Predictive pruning can save computational costs when comparing with cascading. Predictive pruning can solve the problem of overgeneration of virtual terminals in cascading. However, predictive pruning cannot parse as good as cascading for the extra-grammatical input queries. This make predictive pruning suffers a decline on language understanding.

Third, we compare the difference between English and Chinese grammar. We observe there is no significant difference in our results across English and Chinese. We have proved our parsing frame is portable for different

languages.

5.8 Chapter Summary

In this chapter, we presented our investigation of grammar partitioning and parser composition based on English and Chinese ATIS corpora. We developed the ATIS grammar based on training sets. Then we partitioned it into more than 60 sub-grammars and compose these sub-parsers by two parser composition methods which described in Chapter 5. We also use a single GLR parser for unpartitioned grammar as our benchmark. We found that grammar partitioning significantly reduces the sizes of LR(1) parsing tables and is suitable for handling large grammars, when compared with the single parser. For the two different parser composition algorithms, the parsing efficiency of predictive pruning is better than cascading algorithm. However, cascading can produce a greater number of partial parses and thus obtains a better performance on understanding accuracies.

In the next chapter, we will give a brief summary of this thesis and summarize major contributions.

Chapter 6

Conclusions

6.1 Thesis Summary

In this thesis, we have developed a modular parsing framework – grammar partitioning and parser composition, for natural language processing. It aims to handle ambiguity, space and efficiency problems for large grammars, and can also handle extra-grammatical sentences.

As demonstrated in the experiments described in Chapter 5, the techniques of grammar partitioning and parser composition successfully meets these objectives. Our parsing framework is developed based on the GLR parser, which can generate multiple parses for an ambiguous sentence. We used a shortest-path algorithm to find the "best" path from the output LMG to represent the input sentence.

We translated the English ATIS-3 queries in class-A into Cantonese Chinese, as a parallel corpus for our experiments. Our unpartitioned English grammar has 72,869 states in its parsing table, while partitioned English

grammar has 3,350 states. The unpartitioned Chinese grammar has 29,734 states in its parsing table, while partitioned Chinese grammar has 3,894 states. Grammar partitioning greatly economizes on the parsing table sizes. This can save time to construct and space to store a large size of table.

Parser composition imparts robustness for parsing, it can produce partial parses while GLR parser could not. We have applied two techniques for parser composition: (i) parser composition by cascading and (ii) parser composition with predictive pruning. Cascading differs from predictive pruning as it does not constrain the parser where to start. Robustness in parsing led to higher understanding performance for cascading. For ATIS English test set 94, semantic concept error rates are 40.4% for the single GLR, 11.7% for cascading and 29.1% for predictive pruning. For Chinese test set 94, the values are 44.9% for single GLR, 12.7% for cascading and 28.4% for predictive pruning.

6.2 Thesis Contributions

The major contributions of this thesis are the following:

1. Development of a modular parsing framework based on the GLR parser, capable of significantly reducing the size of the LR parsing table for handling large grammars in natural language. Reducing the parsing table size can save time for constructing the table and space for storing the table. Hence if any grammar rule needs to be modified, we only need to regenerate the corresponding subparser's table, instead of the

entire parsing table (for an unpartitioned grammar).

2. Implementation of two parser composition algorithms – parser composition by cascading and parser composition with predictive pruning. We explore the trade-off between language understanding accuracies and computational costs for the two different parser composition algorithms. Predictive pruning is more efficient but obtained a lower understanding accuracy.

3. The demonstration of the language portability, robustness and efficiency of our parsing framework in ATIS domain in terms of the size of the parsing tables, grammar coverage, computational costs, and understanding accuracies. Cascading and predictive pruning greatly increase the percentage of partial parses in the test sets when compared to original GLR parsing algorithm. Our parsing framework is directly portable from English to Chinese.

6.3 Future Work

There are several areas for future work which may further improve the performance of the parsing framework. They are related on grammar partitioning, best path selection and robust parsing.

6.3.1 Statistical Approach on Grammar Partitioning

Grammar partitioning is desirable, but it is not clear how it should be done in general case. Weng et al. [35] proposed statistical guidelines for grammar

partitioning.

Each sub-grammar is a cluster in the entire grammar. First, a weighted graph is created to describe the connections among different clusters. When training data is available, probability weights on the edges of weighted graph can be computed based on the actual calls between the corresponding clusters. This way is related to the grammar chunking approach by Rayner [21].

For better cluster quality, a set of simple heuristics for the refinement of grammar is used. There are two types of refinements, merge to form a bigger clusters, and split to form smaller clusters. The merge operations are:

1. Compute the transitive closures of the calling relations for clusters, and replace the original clusters with their closures.

2. If a cluster has an empty INPUT set, duplicate the cluster for each of its parent clusters, and merge the copies with its parents.

The split operations can be through removing the edges with low weights or the ones that lead to minimum changes with respect to its original graph using *Kullback-Leibler distance.*

6.3.2 Probabilistic modeling for Best Parse Selection

We have reported encouraging results in the ATIS domain in terms of understanding accuracy. However, we need a more general and systematic method to select the best parse from multiple parses.

We plan to use a probabilistic context-free grammar (PCFG) [27][17][37], instead of a CFG. Wright[38] proposed a GLR parser to handle probabilistic

grammars. We can estimate the rule probabilities of a PCFG by parsing a training corpus. The frequencies of rule firing can be determined from the training corpus. Then we can normalize these frequencies into rule probabilities.

6.3.3 Robust Parsing Strategies

Since the parsing efficiency of predictive pruning over cascading is highly desirable, we will continue with the future task of improving performance in parse coverage and language understanding. The incorporation of robust parsing strategies with predictive pruning will be our another future direction as well.

A straight-forward to achieve robust parsing is to allow word skipping to handle several types of extra-grammatical phenomena, such as unknown words, ellipses, redundancy, etc.

In this thesis, we have obtained encouraging parsing results for Chinese and English queries from an ATIS corpus. In the future, we will try to expand our work in these directions.

Bibliography

[1] S. Abney. *Parsing by Chunks*, chapter In Principle-Based Parsing: Computation and Psycholinguistics, pages 257–278. Kluwer Academic Publishers, 1991.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[3] A. Aho and J. Ullman. *The Theory of Parsing, Translation and Compiling*, volume 1 of *Parsing.* Prentice-Hall, Englewood Cliffs, N. J., 1972.

[4] A. Aho and J. Ullman. Principles of compiler designer. In *Computer Science and Information Processing.* Addison-Wesley, 1977.

[5] J. Amtrup. Parallel parsing: Different distribution schemata for charts. In *Proceedings of the 4th International Workshop on Parsing Technologies*, 1995.

[6] R. Basili, M. T. Pazienza, and F. M. Zanzotto. Customizable modular lexicalized parsing. In *Proceedings of 6th International Workshop on Parsing Technologies*, 2000.

[7] J. Earley. *An Efficient Context-free Parsing Algorithm.* PhD thesis, Carnegie Mellon University, 1968.

[8] J. Eisner and G. Satta. Efficient parsing for bilexical context-free grammars and head automation grammars. In *Proceedings of ACL*, 1999.

[9] D. Goddeau. Using probabilistic shift-reduce parsing in speech recognition systems. In *Proceedings of International Conference on Spoken Language Processing*, pages 321–324, October 1992.

[10] F. S. Hillier and G. J. Lieberman. *Introduction to Operations Research.* McGraw-Hill, Inc., 6 edition, 1995.

[11] S.C. Johnson. Yacc-yet another compiler compiler. CSTR 32, Bell Laboratories, Murray Hill, N.J., 1975.

[12] M. Kay. Algorithm schemata and data structures in syntactic processing. CSL-80 12, Xerox PARC, 1980.

[13] D. E. Knuth. On the translation of languages form left to right. *Inform. Contr.*, 8:607–639, December 1965.

[14] A. Korenjak. A practical method for constructing lr(k). *CACM 12*, 11, 1969.

[15] A. Lavie. *GLR*: A Robust Grammar-Focused Parser for Spontaneously Spoken Language.* PhD thesis, Carnegie Mellon University, May 1996.

[16] A. Lavie and M. Tomita. *Recent advances in parsing technology*, volume 1 of *Text, speech and language technology*, chapter *GLR**–An Efficient Noise-Skipping Parsing Algorithm for Context-Free Grammars, pages 183–200. Kluwer Academic Publishers, 1996.

[17] W. J. M. Levelt. Formal grammars in linguistics and psycholinguistics. *Mouton*, 1, 1974.

[18] R. Moore. Improved left-corner chart parsing for large context-free grammars. In *Proceedings of the 6th International Workshop on Parsing Technologies*, 2000.

[19] R. Nozohoor-Farshi. *GLR Parsing for ϵ-Grammars*, chapter 5, pages 61–75. Kluwer Academic Publishers, 1991.

[20] P. Price. Evaluation of spoken language systems: The atis domain. In *Proceedings of the ARPA Human Language Technology Workshop*, pages 91–95, 1990.

[21] M. Rayner and D. Carter. Fast parsing using pruning and a grammar specialization. In *Proceedings of 1996 ACL*, 1996.

[22] S. Seneff. Tina: A probabilistic syntactic parser for speech understanding systems. In *Proceedings of Speech and Natural Language Workshop*, pages 168–178, February 1989.

[23] S. Seneff. Tina: A natural language system for spoken language applications. *Computational Linguistics*, 18(1):61–86, 1992.

[24] P. Shann. *Generalized LR Parsing*, chapter Experiments with GLR and Chart Parsing, pages 17–34. Kluwer Academic Publishers, 1991.

[25] K. C. Siu and H. Meng. Semi-automatic acquisition of domain-specific semantic structures. In *Proceedings of Eurospeech.*, 1999.

[26] S. Steel and A. De Roeck. Bi-directional parsing. In Hallam and Mellish, editors, *Proceedings of the 1987 AISB Conference*, London, 1987. J. Wiley.

[27] P. Suppes. Probabilistic grammars for natural languages. *Synthese*, 22:95–116, 1968.

[28] M. Tomita. *Efficient Parsing for Natural Language.* Kluwer Academic Publishers, Boston, MA, 1985.

[29] 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)*, pages 1569–1572, April 1986.

[30] M. Tomita and See-Kiong Ng. *The Generalized LR Parsing Algorithm*, chapter 1, pages 1–16. Kluwer Academic Publishers, 1991.

[31] E. Ukkonen. Lower bounds on the size of deterministic parsers. *Computer and System Sciences*, 26:153–170, 1983.

[32] W. Ward. Understanding spontaneous speech. In *Proceedings of Speech and Natural Language Workshop*, pages 137–141, February 1989.

[33] W. Ward. The cmu air travel information service: Understanding spontaneous speech. In *Proceedings of Speech and Natural Language Workshop*, pages 127–129, June 1990.

[34] W. Ward. Evaluation of the cmu atis system. In *Proceedings of Speech and Natural Language Workshop*, pages 101–105, 1991.

[35] F. Weng, H. Meng, and P.C. Luk. Parsing a lattice with multiple grammars. In *Proceedings of the 6th International Workshop on Parsing Technologies*, 2000.

[36] F. Weng and A. Stolcke. Partitioning grammar and composing parsers. In *Proceedings of the 4th International Workshop on Parsing Technologies*, September 1995.

[37] C. S. Wetherall. Probabilistic languages: A review and some open questions. *Computing Surveys*, 12:361–379, 1980.

[38] J. H. Wright and E. N. Wrigley. *GLR Parsing with Probability.*, chapter 8, pages 113–128. Kluwer Academic Publishers, 1991.

[39] D. H. Younger. Recognition and parsing of context free languages in time n^3. *Information and Control*, 10:198–208, 1967.

Appendix A

ATIS-3 Grammar

This appendix contains a subset of the English and Chinese ATIS-3 SENTENCE-level rules, high level syntactic rules and low level semantic rules.

A.1 English ATIS-3 Grammar Rules

```
sentence-level rules
====================
S :=AIRLINE_CODE FLIGHT_PP
S :=AIRLINE_CODE TIME_NP FLIGHT_PP
S :=ASK ASK FLIGHT_PP MEAL_PP
S :=ASK FLIGHT_ID
S :=ASK FLIGHT_NP
S :=ASK QUANT AIRLINE_CODE FLIGHTS
S :=ASK QUANT FLIGHT
S :=FLIGHT LOC FLIGHT_PP
S :=FLIGHT_NP AND CLASS_TYPE
S :=PLEASE ASK FLIGHT_NP
S :=PLEASE ASK FLIGHT_NP TIME_NP
S :=PLEASE ASK QUANT FLIGHT_NP
S :=QUEST AIRLINE_CODE
S :=QUEST AND QUEST
S :=QUEST FLIGHT_PP TIME_NP
S :=QUEST TIME_NP
S :=WHAT FLIGHT are FLIGHT_PP
S :=WHAT QUANT FLIGHT_NP
```

```
S :=WHETHER QUANT FLIGHT_NP LOC
S :=which FLIGHT_NP

high level rules
================
FLIGHT_NP :=FLIGHT_NP AND FLIGHT_NP
FLIGHT_NP :=FLIGHT FLIGHT_PP
FLIGHT_NP :=FLIGHT FLIGHT_PP FLIGHT_PP
FLIGHT_NP :=TIME_NP FLIGHT FLIGHT_PP

FLIGHT_PP :=which FLIGHT_PP
FLIGHT_PP :=that FLIGHT_PP
FLIGHT_PP :=ARRIVAL
FLIGHT_PP :=BETWEEN [ROUND_TRIP_REQUIRED]
FLIGHT_PP :=DEPARTURE
FLIGHT_PP :=DEPARTURE [AND] ARRIVAL [TIME_NP]
FLIGHT_PP :=DEPARTURE ARRIVAL [THEN] ARRIVAL
FLIGHT_PP :=STOP

ARRIVAL :=TO LOC
ARRIVAL :=TO TIME_NP
ARRIVAL :=be there by TIME_NP
ARRIVAL :=TO TIME_NP in LOC

DEPARTURE :=FROM LOC TIME_NP
DEPARTURE :=FROM [either] LOC
DEPARTURE :=FROM TIME_NP

LOC :=CITY_NAME [STATE_NAME]
LOC :=AIRPORT_CODE
LOC :=CITY_NAME COUNTRY_NAME
LOC :=LOC OR LOC
LOC :=anywhere

TIME_NP :=[on] DATE
TIME_NP :=UPDATE
TIME_NP :=[on] DATE TIMERANGE
TIME_NP :=TIMERANGE [on] DATE
TIME_NP :=TIME of DATE

TIMERANGE :=ABOUT TIME
```

```
TIMERANGE :=TIME TIME_TO TIME
TIMERANGE :=ABOUT TIME TIME_TO TIME
TIMERANGE :=BEFORE TIME

QUEST :=WHICH OBJ
QUEST :=WHICH FLIGHTS
QUEST :=WHICH_TYPE FLIGHT
QUEST :=HOW_MANY FLIGHT_NP

low level rules
===============
TO :=into
TO :=fly to
TO :=arrive to
TO :=arrive
TO :=arrives

FROM :=from
FROM :=departing from
FROM :=depart from
FROM :=leave from
FROM :=leaving

ASK :=may i
ASK :=need to
ASK :=want to
ASK :=like to
ASK :=would like to

DAY_NAME :=yesterday
DAY_NAME :=tuesday
DAY_NAME :=wednesday
DAY_NAME :=friday's
DAY_NAME :=sundays

CITY_NAME :=westchester
CITY_NAME :=atlanta
CITY_NAME :=chicago
CITY_NAME :=milwaukee

APM :=morning
```

```
APM :=afternoon
APM :=evening
APM :=day

WHETHER :=does
WHETHER :=is there
WHETHER :=do
WHETHER :=are they
```

A.2 Chinese ATIS-3 Grammar Rules

```
sentence-level rules
====================
S :=AIRLINE QUEST
S :=AIRLINE_CODE 既 QUEST
S :=ASK FLIGHT_NP
S :=ASK FLIGHT_PP
S :=ASK QUANT FLIGHT_NP
S :=ASK QUEST
S :=ASK TIME_NP QUEST
S :=ASK 係 FLIGHT_PP
S :=FLIGHT_NP HOW_MUCH
S :=FLIGHT_PP 既 FLIGHT_NUMBER FLIGHT
S :=FLIGHT_PP 要 QUEST
S :=HOW_MUCH 係 AIRLINE
S :=HOW_MUCH 係 AIRLINE AND AIRLINE
S :=PLEASE ASK ASK FLIGHT_NP
S :=PLEASE ASK FLIGHT_NP AND OBJ
S :=PLEASE ASK FLIGHT_PP TIME_NP 又 FLIGHTS
S :=PLEASE ASK QUANT TIME_NP OR FLIGHT_NP
S :=PLEASE ASK QUEST 有 FLIGHT_NP
S :=PLEASE ASK TIME_NP FLIGHT_NP
S :=PLEASE ASK TIME_NP QUEST
S :=TIME_NP QUEST MEAL_DESCRIPTION
S :=WHETHER QUANT FLIGHT_NP

high level rules
================
FLIGHT_NP :=[會] [係] FLIGHT_PP [既] FLIGHT
```

```
FLIGHT_NP :=FLIGHT_PP 同 時 FLIGHT_PP [既] FLIGHT
FLIGHT_NP :=FLIGHT [係] FLIGHT_PP
FLIGHT_NP :=FLIGHT_PP AND FLIGHT_PP [既] FLIGHT

FLIGHT_PP :=ARRIVAL
FLIGHT_PP :=ARR_DEPART
FLIGHT_PP :=BETWEEN
FLIGHT_PP :=DEPARTURE
FLIGHT_PP :=DEPARTURE ARRIVAL
FLIGHT_PP :=DEPARTURE ARRIVAL [THEN] ARRIVAL

ARRIVAL :=[係] LOC 降 落
ARRIVAL :=[係] TIME_NP [係] LOC 降 落
ARRIVAL :=[係] TIME_NP 到 LOC
ARRIVAL :=[係] TIME_NP 到 達 LOC

DEPARTURE :=[TIME_NP] 由 LOC
DEPARTURE :=[TIME_NP] 從 LOC
DEPARTURE :=[TIME_NP] 係 LOC 起 飛
DEPARTURE :=TIME 機 去
DEPARTURE :=TIME 機 返

TIME_NP :=DATE TIME
TIME_NP :=DAY_NAME TIME
TIME_NP :=FLIGHT_DAYS TIME
TIME_NP :=WEEK TIMERANGE

TIMERANGE :=ABOUT TIME TIME_TO TIME
TIMERANGE :=TIME TIME_TO TIME
TIMERANGE :=[ABOUT] TIME 左 右
TIMERANGE :=TIME BEFORE

QUEST :=HOW_LONG
QUEST :=HOW_MUCH OBJ
QUEST :=WHICH OBJ

low level rules
===============
ASK :=我 想 搵
ASK :=話 比 我 知
ASK :=話 我 知
```

```
ASK :=請 問

DAY_NAME :=星 期 一
DAY_NAME :=工 作 天
DAY_NAME :=禮 拜 一
DAY_NAME :=星 期 三

AIRLINE_CODE :=三 角 州
AIRLINE_CODE :=加 拿 大 國 際 航 空
AIRLINE_CODE :=聯 合 航 空 公 司
AIRLINE_CODE :=美 國 航 空 公 司

APM :=上 晝
APM :=下 晝
APM :=上 午
APM :=下 午

TIME_TO :=至
TIME_TO :=至 到

WHETHER :=係 唔 係
WHETHER :=是 不 是
WHETHER :=有 冇
WHETHER :=可 唔 可 以
```