Using Dominator-Modifier Relations to Disambiguate A Sentence

Takanori Tsukada

AI TR87-60 August, 1987
USING DOMINATOR-MODIFIER RELATIONS TO DISAMBIGUATE A SENTENCE

by

TAKANORI TSUKADA, B.S.

THESIS
Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCES

THE UNIVERSITY OF TEXAS AT AUSTIN
August, 1987
Acknowledgments

I gratefully acknowledge the encouragement, support and valuable suggestions of my supervisor, Dr. Robert F. Simmons. I would also like to thank Dr. Vipin Kumar for his careful reading and helpful suggestions. I am grateful to my friends who encouraged me in many ways. Special thanks are due to Luther Karl Branting, Jungyun Seo, Olivier Winghart and Wing Kong Wong. They helped me correct English and gave me many useful comments. Finally, I express my gratitude to my supervisors and colleagues at Hitachi Software Engineering Co., Ltd. for giving me the opportunity to study at the University of Texas at Austin.

TAKANORI TSUKADA

The University of Texas at Austin
August, 1987
Abstract

This work introduces a system to parse and disambiguate ambiguous sentences efficiently. The system consists of three parts: a compiler, a parser, and a disambiguation program. Dominator-modifier relations, which are edges between two words, are produced to represent an ambiguous sentence. The relations are calculated using an efficient parsing method. They represent the ambiguities of a sentence clearly and facilitate ambiguity detection. These pairs and tree structures are produced by the parser and are used later in a disambiguation program. The parser uses a bottom-up parsing algorithm, the LR(0) algorithm with dynamic programming. A compiler that translates Definite Clause Grammar rules into subprograms of the parser is also described. A disambiguation program is described to show how dominator-modifier relations are used for disambiguation by heuristics. This experiment shows that dominator-modifier relations make it easier to solve lexical and structural ambiguities.
# Table of Contents

Acknowledgments

Abstract

Table of Contents

List of Tables

List of Figures

1. Introduction
   1.1 Problems in Current Parsing Systems
   1.2 Proposed Solution
   1.3 Contents of this Thesis

2. Background
   2.1 Unification Grammars
   2.2 DCG
   2.3 Parsing Algorithms
   2.4 Parsing Complex-Feature-Based Grammars
      2.4.1 Complexity of parsing
      2.4.2 Reducing Complexity
      2.4.3 Feature and Heuristic Search
      2.4.4 Restricting Unification Grammars
      2.4.5 Restricting Feature Structures
2.5 Disambiguation ........................................... 23
2.6 Using LR Parsing in NLP ............................... 24
   2.6.1 Modifying LR Parsing for NLP ................... 24
   2.6.2 Conflicts ........................................... 25
   2.6.3 Tomita’s Algorithm ............................... 26
2.7 Toward a Practical System ............................. 29

3. Theoretical Basis ......................................... 30
   3.1 X-Theory ............................................. 30
   3.2 Head, Dominator, and Modifier .................... 32
   3.3 Ambiguity ............................................ 34
   3.4 mod Relations and Ambiguities .................... 36
      3.4.1 mod Relation and Structural Ambiguities ....... 36
      3.4.2 mod Relations and Word Usage Ambiguities ..... 40
      3.4.3 Using Functional Relations in mod Relations ... 41
      3.4.4 mod Relations and Spurious Ambiguities ....... 44
      3.4.5 Number of mod Relations ....................... 46
      3.4.6 Ambiguities in Terms of mod Relations ......... 47
      3.4.7 Final Remarks on the mod Relation ............. 50
   3.5 General Strategies of Disambiguation ............... 50

4. Implementation ........................................... 53
   4.1 System Configuration ................................ 53
   4.2 Compiler ............................................ 53
   4.3 Grammar and Dictionary Format ..................... 56
   4.4 Parser .............................................. 59
4.4.1 Configuration .................................................. 59
4.4.2 Output of LRPARSER ........................................ 61
4.4.3 LR(0) Parser .................................................. 62
4.4.4 An Alternative LR(0) Parsing Table ...................... 64
4.4.5 Stack .......................................................... 65
4.4.6 Compiled Grammar Format .................................. 67
4.4.7 Disambiguation Program .................................... 70
4.5 Debugger .......................................................... 72
4.6 Experimental Results ........................................... 74

5. Conclusions .......................................................... 78
5.1 Conclusions ....................................................... 78
5.2 Future Work ....................................................... 78

A. Parsing Algorithms .................................................. 80
   A.1 The CKY Algorithm ............................................. 80
   A.2 Earley’s Algorithm ............................................. 80
   A.3 LR Parsing ....................................................... 83

B. Derivation of the Complexity of Parsing ....................... 88

C. Examples of Data Structures ....................................... 93
   C.1 An Example Grammar in DCG ................................. 93
   C.2 An Example Grammar in CFG ................................. 94
   C.3 Output of the Parser ........................................... 94
      C.3.1 Net Output of LRPARSER ................................. 94
      C.3.2 Retracted Output of LRPARSER ......................... 97
C.4 An Alternative LR(0) Parsing Table ................. 98
  C.4.1 from/1 and goto/3 .......................... 98
  C.4.2 setofitem/2 ................................. 99
C.5 Stack .............................................. 100
C.6 Compiled Grammar ................................. 101
  C.6.1 gr1/8 ..................................... 101
  C.6.2 gr1/6 ..................................... 103

BIBLIOGRAPHY .......................... 105

Vita
List of Tables

2.1 Ambiguity and the Number of Structures  

2.2 LR Parsing Table with Conflicts  

2.3 Two Sequences of Stack and Input  

3.1 mod Relations in Five Readings  

3.2 Number of mod Relations  

3.3 Example Grouping of mod Relations  

4.1 Experimental Result  

A.1 LR Parsing Table  

A.2 Sequence of Stack and Input  

B.1 Equations for CFG  

B.2 Equations for Complex-Feature-Based Grammar
List of Figures

2.1 Feature Structures in DCG and Other Unification Grammars ........................................ 6
2.2 DCG Rules and Translated Prolog Predicates ............................................................... 7
2.3 Example Execution of DCG ......................................................................................... 9
2.4 Example Grammar ....................................................................................................... 13
2.5 Complex-Feature-Based Grammar ............................................................................... 13
2.6 Complex Feature Structures ....................................................................................... 14
2.7 Plane of Parsing Methods ............................................................................................ 20
2.8 Locality on Information Flow ....................................................................................... 21
2.9 Graph-Structured Stack ............................................................................................... 27

3.1 Grammar Based on \( \bar{X} \)-Theory ........................................................................ 31
3.2 Syntax Structure Based on \( \bar{X} \)-Theory ................................................................. 31
3.3 Generalized Grammar Based on \( \bar{X} \)-Theory ......................................................... 31
3.4 Example Grammar ...................................................................................................... 33
3.5 A Parse Tree and Heads .............................................................................................. 34
3.6 Ambiguous Grammar .................................................................................................. 35
3.7 Two Readings for "It is hard for him to read the book." ......................................... 36
3.8 Five Syntactic Structures ........................................................................................... 38
3.9 Five Syntactic Structures (Continued) ....................................................................... 39
Chapter 1

Introduction

1.1 Problems in Current Parsing Systems

Unification grammars are currently a trend in linguistics and natural language processing (NLP). However, it is difficult to apply unification grammars to parsing natural languages because of ambiguities in sentences. The number of readings for a sentence grows exponentially as the sentence becomes longer.

A method is known that prevents a parser from generating inappropriate readings, but this method is not flexible enough to disambiguate a large class of English sentences. It eliminates all possible readings in some cases, while allowing all readings in some other cases.

Another problem with unification grammars is that in the worst case when the parser fails to disambiguate a sentence, the parsing task takes exponential time in regard to the length of an input sentence. This is because, unlike in the case of context-free grammars (CFG's), the parser constructs exponentially many structures regardless of what parsing method it uses. Therefore, we need to avoid applying complex-feature-based grammars directly to parsing.
1.2 Proposed Solution

One way of reducing complexity of parsing is to restrict the features used during parsing so that the number of structures does not grow exponentially. The features should also be useful for the parser to disambiguate the sentence to reduce the number of readings. We propose the head of a phrase as such a feature. We also propose the dominator-modifier relation between the heads of phrases as the information used to disambiguate a sentence.

Tomita uses a relation called an explanation template to disambiguate a sentence by questioning to a user [Tomita 86]. An explanation template is similar to the dominator-modifier relation, but is not based on linguistic theory. In this thesis, we explore a relation based on X-Theory\textsuperscript{1} in linguistics and analyze the complexity of parsing when the head of each phrase is used as a feature. We also implement a disambiguation routine.

Since our goal is to build an efficient parsing system, we need to choose a fast parsing algorithm. We choose Tomita’s algorithm, which is one of the fastest parsing algorithm for CFG’s.

1.3 Contents of this Thesis

In Chapter 2, first we will explain unification grammars and the problems that arise when parsing a sentence using an unification grammar, particularly a complex-feature-based grammar. Then we will describe Tomita’s algorithm. Finally we summarize the important properties for parsing systems.

\textsuperscript{1}Pronounced X-bar Theory.
In Chapter 3, first we will explain X-Theory in linguistics to introduce the dominator, the modifier, and the head of a phrase. Then we will show how the dominator-modifier relation between heads is used to represent ambiguous sentences and to solve ambiguities. We will explore several different alternatives for the dominator-modifier relation. We will also analyze the parsing complexity when the head is used as the feature.

In Chapter 4, first the overall structure of the system will be described. Then the data structure and detail of three parts of the system — the compilers, the parser including an example of disambiguation routines, and the debugger — will be set forth.
Chapter 2

Background

In this chapter, first we will explain unification grammars and the problems that arise when parsing a sentence using an unification grammar, particularly a complex-feature-based grammar. Then we will describe Tomita’s algorithm. Finally we summarize the important properties for parsing systems.

2.1 Unification Grammars

Grammar formalisms based on a unification operation are currently a trend in both linguistics and NLP. Such grammars are called unification grammars [Shieber 86]. Many unification grammars restrict their syntactic structure within the class of CFG’s, while using complex features. Unification-based formalisms are:

- surface-based, thus non-transformational
- informational: information is associated with string
- inductive: information is combined in bottom-up fashion
- declarative, as opposed to procedural

To satisfy these requirements, these formalisms use a set of feature-value pairs associated with each string, and are thus called complex-feature-based. We refer to these pairs as feature structures. In unification grammar formalisms,
the combining operation over feature structures, or *unification*, plays a central role. Examples of unification grammars are Lexical Functional Grammar (LFG) [Kaplan 83], Generalized Phrase Structure Grammar (GPSG) [Gazdar 85], Functional Unification Grammar (FUG) [Kay 84] and Combinatory Categorial Grammar (CCG) [Wittenburg 86]. Sells gives an introduction to LFG and GPSG in [Sells 85]. Definite Clause Grammar (DCG) [Pereira 80] can be considered as a special case of unification grammars. There are some systems, such as DCG and PATR-II, that parse sentences directly by using a unification grammar [Shieber 84]. But for processing consideration, we doubt that it is necessary or preferable to use a unification grammar directly in parsing.

### 2.2 DCG

To illustrate unification grammars, let us explain one of them, DCG. The difference between CFG and DCG is that in DCG:

- Nonterminals may have additional arguments, just as Prolog predicates have.

- Arbitrary Prolog program can be written in the right side of grammar rules. They are closed in braces ({ }).

DCG is considered a special case of unification grammar because its definition of unification is different from that used in other unification grammars. Also in DCG, a feature structure is called a *term*. A term is not an unordered set, but an ordered list of values. A value is identified not by a feature associated with it but by its linear order in the term. *Arity*, the number of elements in a term, is significant in unification. The feature of each
DCG feature structure 1: \([value_1, value_2, \ldots]\)

DCG feature structure 2: \([value_2, value_1, \ldots]\)

Other's feature structure 1: \[
\begin{array}{l}
\text{feature}_1: value_1 \\
\text{feature}_2: value_2 \\
\end{array}
\]

Other's feature structure 2: \[
\begin{array}{l}
\text{feature}_2: value_2 \\
\text{feature}_1: value_1 \\
\end{array}
\]

Figure 2.1: Feature Structures in DCG and Other Unification Grammars

value is implicitly defined by its position in a list or a predicate. In Figure 2.1, the first two lines represent structures used in DCG, while the next two are those used in other unification grammars. The first two are not identical, while the other two are. However, there is research to include unordered sets of feature-value pairs in DCG [Mukai 85a] [Mukai 85b].

Another characteristic of DCG is that it is directly executed as a Prolog program. DCG rules are translated into Prolog program by a preprocessor built in Prolog, and the program is executed as a top-down parser. To illustrate the translation, a part of a grammar is shown in Figure 2.2. The first two are corresponding DCG rules and the last two are Prolog predicates translated from the DCG rules, which correspond to the following CFG rules:

\[
\begin{align*}
\text{Sentence} & \rightarrow \text{NP} \; \text{VP} \\
\text{NP} & \rightarrow \text{time}
\end{align*}
\]

The first DCG rule is considered to show the following:

- sentence is made by nonterminals np and vp.
sentence(S_frame,s(NP,VP)) -->
   np(NP_frame,NP),
   {case(NP_frame,nominative)},
   vp(VP_frame,VP),
   {make_new_frame(NP_frame,VP_frame,S_frame)}.

np([pred=time,number=singular,person=3],[pred=time]) -->
   [time].

sentence(S_frame,s(NP,VP),SO,S) :-
   np(NP_frame,NP,SO,S1),
   case(NP_frame,nominative),
   vp(VP_frame,VP,S1,S),
   make_new_frame(NP_frame,VP_frame,S_frame).

np([pred=time,number=singular,person=1],
   np(time),[time|S],S).

Figure 2.2: DCG Rules and Translated Prolog Predicates

- np has nominative case.
- Whether it is syntactically and semantically correct to combine np and
  vp is checked by using information carried by variables NP_frame and
  VP_frame.

make_new_frame/3\(^1\) is a Prolog predicate checking this attachment. It must
be defined by a user. The first argument of sentence/4, S_frame, contains
semantic information. The second argument, s(NP,VP), contain the syntac-
tic structure. The Prolog translator adds two arguments to each predicate,
except ones enclosed in braces. The third argument, SO, indicates the position
of the left end of the nonterminal. The fourth argument, S, indicates
the position of the right end of the nonterminal. More precisely, the third

\(^1\)x/n denotes a Prolog predicate x with n arguments
argument is the remaining part of the sentence that the parser has not parsed before it parses the nonterminal. The fourth argument is the remaining part of the sentence after the parser parses the nonterminal.

The second DCG rule represents the following:

- NP is made by a terminal time. Terminals are closed in brackets ([ ]).
- The meaning of time is time, the number of time is singular, the person of time is third.

The translated Prolog predicates are called by the user as in Figure 2.3. The variables S_frame and Structure will be instantiated so that S_frame becomes the semantic structure and Structure becomes the syntactic structure.

Prolog is often used to implement NLP systems. Translated programs are interpreted by Prolog or compiled into efficient code. No special parser is necessary. An attractive feature of DCG is that a grammar is declaratively described and also directly executed as a Prolog program. This is one of the reasons why Prolog is appropriate in NLP. However, DCGs with left recursion are unusable because DCG parsing is top-down. Despite this inconvenience, DCGs are often used because of the advantages described above. Moreover, left-recursive rules can be systematically transformed into non-left-recursive rules by an algorithm described in [Aho 86].

2.3 Parsing Algorithms

One of the advantages of using CFG in unification grammars is that many efficient parsing algorithms are available because it is well studied in automata theory. Aho and Ullman discuss many parsing methods in detail
On Prolog, a user asks:

?- sentence(S_frame, Structure, [time, flies, like, an, arrow], []).

After parsing, Prolog answers:

S_frame = [subj=[pred=time, number=singular, person=3],
           v=[pred=fly, number=singular, person=3],
           adjunct=[p=like, obj=[pred=arrow, number=singular, person=3, det=indefinite]]]

Structure = s(np(time),
              vp(v(flies),
              pp(p(like),
              np(det(an),
              n(arrow)))))

Figure 2.3: Example Execution of DCG

[Aho 72]. There are three general types of parsers for CFG. Universal parsing methods such as the Cocke-Kasami-Younger (CKY\textsuperscript{2}) and Earley's algorithm can parse any CFG. The CKY algorithm and Earley's algorithm are shown in Appendix A.1 and Appendix A.2. These bottom-up algorithms are also called tabular methods because of their data structures. The CKY algorithm uses the technique of a dynamic programming. The CKY algorithm is the same as one of the bottom-up chart-parsing algorithms [Kay 80], which is well known in NLP. Earley's algorithm is the same as an active chart-parsing algorithm.

\textsuperscript{2}Also called CYK algorithm.
Other methods are classified as either top-down or bottom-up. The general form of top-down parsing, called recursive-descent parsing and can only be used for grammars without left recursion. It works without backtracking only for some classes of CFG. For example, LL\((k)\), a top-down parsing algorithm, works without backtracking on a grammar class called LL\((k)\). The first “L” is for left-to-right scanning of the input, the second “L” is for constructing a left-most derivation in reverse, and \(k\) is the number of input symbols that the parser looks ahead in making parsing decisions. Bottom-up parsing algorithms such as LR\((k)\) work without backtracking on larger classes of grammar. The “L” and \(k\) are the same as those of LL\((k)\). The “R” is for constructing a right-most derivation in reverse. Construction of a parsing table is necessary to use LR parsing. An example of a table is shown in Appendix A.3. The parsing table is different from table in tabular methods. One of the LR(1) methods, LALR (look-ahead LR) parsing, is commonly used by compilers to parse programming languages [Aho 86]. LALR parsing is also used in some research in NLP [Shieber 83].

In NLP research, other parsing algorithms that are easy to debug are often used. The most popular is DCG. DCG corresponds to LL\((0)\). Left-corner parsing, a bottom-up parsing method similar to LR\((0)\), is another method that is often used. Although in left-corner parsing it is necessary to translate DCG grammar rules into left-corner parser, this can be done systematically by a program [Matsumoto 83]. The translated program can be interpreted directly by Prolog or compiled into efficient code. No table or special parser is necessary, which is one of the reasons to use left-corner parsing.

Natural languages are surely not in the grammar classes LL\((k)\) and
LR(k) for small k. Thus LL algorithms, including DCG and LR parsing, need backtracking. But backtracking is still not very efficient for parsing long sentences using ambiguous grammars; tabular methods are preferred in such cases. Backtracking methods take $O(c^n)$ time\(^3\) for a sentence of length $n$, where $c$ is some constant. On the other hand, the CKY algorithm and Earley’s algorithm take $O(n^3)$ time and are known to work best in practice. There are two algorithms that take $O(n^{2.81})$ time and $O(n^{3 + \frac{3}{\log n}})$ time, but their constant factors are too large to be used practically. See [Hopcroft 79].

According to [Matsumoto 83], left-corner parsing that incorporates top-down prediction by recording partial results and failures eliminates unnecessary calculations caused by backtracking and reduces the parsing time by the order of nearly 10. There is also research to combine LALR and dynamic programming, thus eliminating backtracking [Tomita 85] [Tomita 86].

2.4 Parsing Complex-Feature-Based Grammars

In this section we will discuss the difference of complexity between parsing CFG and complex-feature-based grammars.

2.4.1 Complexity of parsing

The complexity of parsing complex-feature-based grammars is very different from that of CFG’s. Grammar formalisms based on complex features can be considered as generalized CFG’s so that nonterminal symbols are no longer in a finite domain of atoms, but are possibly in the domain of an infinite number of directed graphs. In an infinite nonterminal domain,

\(^3\) $O(z)$ denotes the order of $z$. 
standard parsing methods may no longer be applicable. For instance, preprocessing of grammars for LR and left-corner parsing may fail to terminate. Algorithms performing top-down predictions also may not terminate at parse time. Shieber gives an example of nontermination caused by removal of the context-free base from a grammar, which shows that the context-free base should therefore not be removed from a grammar [Shieber 85].

Even though the recognition problems of LFG and some classes of DCGs and PATR-II grammars are decidable, it has been shown that they are still NP-complete, thus intractable. This means that even though a tabular method such as the CKY algorithm is applied to unification grammars, a parsing takes not \( O(n^3) \) but \( O(c^n) \) time. The reason is as follows: Though parsing in LFG itself is similar to that in CFG's, the parser has to satisfy not only phrase structure rules, but also feature structures of nodes in a tree structure.

Suppose that each reading of a sentence is represented by a tree consisting of nodes and edges. In unification grammar formalisms such as LFG, feature structures are usually built for all partial trees. Such structures contain all the information of the corresponding strings. Thus the structures tend to be large. Moreover, the number of structures will grow exponentially as an input sentence becomes longer because of ambiguities.

To illustrate the difference in complexity between CFG's and complex-feature-based grammars, the number of ambiguities and those of structures built during parsing are calculated for the sentences

\[
I \text{ saw a boy (with a girl)} .^{n-1} \text{ for } n = 1, 2, \ldots
\]

The CKY algorithm is used for both grammars. An ambiguous CFG is shown in Figure 2.4. The complex-feature-based grammar in DCG style is shown in
$S \rightarrow NP \ VP$
$NP \rightarrow Det \ N$
$NP \rightarrow NP \ PP$
$PP \rightarrow P \ NP$
$VP \rightarrow V \ NP$
$VP \rightarrow VP \ PP$

Det $\rightarrow a$
NP $\rightarrow I$
N $\rightarrow boy$
N $\rightarrow girl$
P $\rightarrow with$
V $\rightarrow saw$

Figure 2.4: Example Grammar

\[
s([\text{subj}=\text{NP}|\text{VP}]) \rightarrow \text{np}(\text{NP}), \text{vp}(\text{VP}).
\]
\[
\text{np}([\text{det}=\text{Det}|\text{NP}]) \rightarrow \text{det}(\text{Det}), \text{n}(\text{NP}).
\]
\[
\text{np}([\text{pp}=\text{PP}|\text{NP}]) \rightarrow \text{np}(\text{NP}), \text{pp}(\text{PP}).
\]
\[
\text{pp}([\text{obj}=\text{NP}|\text{P}]) \rightarrow \text{p}(\text{P}), \text{np}(\text{NP}).
\]
\[
\text{vp}([\text{obj}=\text{NP}|\text{V}]) \rightarrow \text{v}(\text{V}), \text{np}(\text{NP}).
\]
\[
\text{vp}([\text{pp}=\text{PP}|\text{VP}]) \rightarrow \text{vp}(\text{VP}), \text{pp}(\text{PP}).
\]
\[
\text{det}([\text{pred}=a]) \rightarrow [a].
\]
\[
\text{np}([\text{pred}=\text{I'}]) \rightarrow ['I'].
\]
\[
\text{n}([\text{pred}=\text{boy}]) \rightarrow [\text{boy}].
\]
\[
\text{n}([\text{pred}=\text{girl}]) \rightarrow [\text{girl}].
\]
\[
\text{p}([\text{pred}=\text{with}]) \rightarrow [\text{with}].
\]
\[
\text{v}([\text{pred}=\text{saw}]) \rightarrow [\text{saw}].
\]

Figure 2.5: Complex-Feature-Based Grammar

Figure 2.5. Here DCG is used to describe the grammar, but the CKY algorithm is used to parse the sentences. All feature structures for the sentence "I saw a boy with a girl" are shown in Figure 2.6.

To analyze the exponential growth of the number of structures, we need to explain local ambiguities first. We say that there is a local ambiguity if there are two or more nodes with a common root nonterminal and common leaf terminals, but with different tree structures.

Let us examine vp nodes over "saw a boy with a girl," and s nodes over "I saw a boy with a girl." The two vp nodes in Figure 2.6 are a result of
Figure 2.6: Complex Feature Structures
Table 2.1: Ambiguity and the Number of Structures

<table>
<thead>
<tr>
<th>n</th>
<th>length of sentence</th>
<th>total number of readings</th>
<th>CFG number of rules appl.</th>
<th>number of structures</th>
<th>complex feature number of structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>2</td>
<td>16</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>5</td>
<td>29</td>
<td>25</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>14</td>
<td>47</td>
<td>37</td>
<td>80</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>42</td>
<td>71</td>
<td>51</td>
<td>199</td>
</tr>
<tr>
<td>6</td>
<td>19</td>
<td>132</td>
<td>102</td>
<td>67</td>
<td>564</td>
</tr>
<tr>
<td>7</td>
<td>22</td>
<td>429</td>
<td>141</td>
<td>85</td>
<td>1687</td>
</tr>
</tbody>
</table>

a local ambiguity over "saw a boy with a girl." The first structure describes the case that "a boy" was "with a girl." The second structure describes that "saw" was done "with a girl." In case of complex-feature-based grammars, local ambiguities affect parsing higher levels in a tree. Therefore, there are two s nodes because of local ambiguity. But in the case of CFG, local ambiguities do not affect parsing higher levels. Only one VP is recorded in the parsing table, as in Figure A.2, and the parser treats the two locally ambiguous VPs as one VP when it builds higher level nodes. Therefore, the parser applies the rule S → NP VP only once for the VP and also builds only one S.

The number of rule applications and structures during parsing are shown in Table 2.1. A derivation of these numbers is shown in Appendix B. Since in complex-feature-based grammar every rule application generates a unique feature structure, the number of rule applications is the same as that of structures.

The number of rule applications is $O(n^3)$ for CFG, $O(n^2)$ for structures, and $O(c^n)$ for complex-feature-based grammars. These results can be
analyzed further. In case of CFG, only categories are recorded for each string. For example, for the whole sentence, only one structure is recorded for S, and this represents the 429 parse trees when $n = 7$. But in the case of complex-feature-based grammar, two partial trees for a string with the same category but with different syntactic structures are recorded separately. Thus for the whole sentence, 429 structures are recorded.

2.4.2 Reducing Complexity

As was shown in Section 2.4.1, the efficiency of parsing depends on how well the parser can avoid generating intermediate structures. There are two strategies to make the parsing of complex-feature-based grammar tractable. In the parsing example in Section 2.4.1, all possible parses are enumerated. Thus, one strategy is to avoid enumeration. The other is to change the representation of feature structures so that the total number grows not exponentially but polynomially. These two strategies are further divided into five.

1. Avoiding enumeration by:
   
   (a) feature checking before applying each rule to reject some rule applications
   
   (b) heuristic search

2. Changing the representation of structures by:

   (a) structure sharing
   
   (b) procrastination
   
   (c) restricting the number of features
In unification grammars, a parser has mechanisms to check syntactic and semantic features during parsing to avoid inappropriate rule applications. As explained in Section 2.2, in DCG system a user can specify the semantic checkings.

But it is not easy to define checking. In a sentence “I saw a man on a hill with a telescope,” it is not always possible to decide whether “with a telescope” is attached to “a hill” or not. The question to be asked is which phrase “with a telescope” is attached to, “saw,” “a man,” or “the hill.” So the criterion of disambiguation should not be an absolute yes or no question, but relative preference. The judgment will vary if the rest of the sentence is different. For example, if the sentence is “A hill with a telescope is at the back of the town,” then “with a telescope” must attach to “a hill” regardless of the appropriateness of the attachment. Thus, it is hard to implement such a flexible judgement by only looking at one attachment “on a hill” and “with a telescope.” Inflexible judgement may reject all the possible attachments in some cases, while it may accept all the possible attachments in other cases. If it accepts all the possible attachments, then the task will be the same as enumerating all possibilities.

*Structure sharing* is a commonly used technique in implementing unification systems, such as Prolog and PATR-II. In complex-feature-based grammars, feature structures are derived by an incremental refinement of feature structures. In a naïve implementation, a new structure is built by copying older ones and then combining the copies according to the constraints stated in a grammar rule. Structure-sharing was designed to eliminate most of such copying. Practical tests suggest that the use of this technique reduces parsing time by as much as 60% [Pereira 85a]. But even though every node
with the same structure is shared, we still need exponentially many sentence nodes. Thus, though structure sharing is an essential technique, it is not efficient enough to reduce the complexity. Also, structure-sharing is more a problem of implementing a programming language than of implementing a parsing system. Therefore, though we will use the technique, we will not discuss it further.

If we see a parsing task as a search problem, depth-first search corresponds to parsing methods with backtracking, such as DCG or left-corner parsing. Graph search provides a more flexible control method that does not need backtracking. One of the uninformed graph search methods is breadth-first search. Breadth-first search corresponds to tabular parsing methods without backtracking, such as the CKY and Earley's algorithm. Those algorithms enumerate all possibilities. An efficient graph search method used in artificial intelligence problems that eliminates unnecessary search by heuristics is called heuristic search, or best-first search [Nilsson 80]. A parser builds only the structures that are likely to contribute to the correct reading. Heuristics are implemented by an evaluation function, which estimates, based on a feature structure, how promising each candidate of an ambiguous syntactic structure is. There are some studies on applications of heuristic graph search methods to parse natural languages [Wittenburg 86]. Application of heuristic search to parsing is promising, but we will try another approach in this thesis.

*Procrastination* is a technique to delay the decision of syntactic structure until enough information is available, typically until semantic processing [Church 82] [Rich 87]. For example, consecutive prepositional phrases (PPs) are represented by a list of PPs without any commitment about their attachment. Fourteen readings of the sentence "I wrote a book for an intro-
duction to the ideas behind parsing," are represented by the list:

\[
[ [ I ]_{sp}, [ wrote ]_{v}, [ a book ]_{pp}, [ with an introduction ]_{pp},
[ to the ideas ]_{pp}, [ behind parsing ]_{pp} ]
\]

Consecutive nouns "run-time parameter optimization" are also represented as the list:

\[
[ [ run-time ]_{s}, [ parameter ]_{s}, [ optimization ]_{s} ]
\]

This list represents two structures. In one, "run-time" modifies "parameter." In the other, "run-time" modifies "optimization." The advantage of procrastination is that if it were possible to procrastinate on all ambiguities, a sentence could be parsed in \(O(n)\) time using a regular grammar.

But there are some disadvantages. Procrastination can only be applied to particular constructs, such as consecutive PPs and consecutive nouns. It is not clear how to procrastinate a combination of different categories, such as a combination of two PPs and a relative clause: "the hardware of memory within the brain whose activity and charges constitute ...." It is also impossible to incrementally disambiguate a sentence, such as choosing three possible attachments from a total of fourteen. We need a more general structure for representation.

2.4.3 Feature and Heuristic Search

Let us now consider the relation between grammar and heuristic search. We may map various parsing methods onto a plane with two axes as shown in Figure 2.7. The horizontal axis corresponds to various control methods. The vertical axis corresponds to various data representations. The left
extreme is breadth-first search, which enumerates all possible readings. The right extreme is best-first search, which finds only one reading. The upper extreme is CFG’s, which use only category for parsing. The lower extreme is complex-feature-based grammars. As we move on the horizontal axis to the right on the plane, the parser spends less time because it builds fewer structures. On the other hand, the overhead to calculate an evaluation function grows. As we move up on the vertical axis, the time complexity of parsing is reduced from $O(c^n)$ to $O(n^3)$. On the other hand, since less knowledge about features is available, it becomes harder to find enough heuristics for heuristic graph search. Breadth-first searches such as the CKY parsers using complex-feature-based grammar perform worst. We do not know what point on the plane corresponds to the best method.
2.4.4 Restricting Unification Grammars

Before looking at the last solution, restricting the number of features, let us look at related proposals, restricting unification grammars. As described in Section 2.4.1, the context-free base should not be removed when extending parsing algorithms. There are some other arguments that restrictions of unification grammar are necessary to achieve efficiency in parsing.

As Pereira points out, the intractability of complex-feature-based grammar is due to lack of \textit{locality} of the flow of information [Pereira 87]. He suggests that we need restrictions on the length of reference paths so that information is passed through the tree only locally. For example, in Figure 2.8, it is unlikely that we need detailed information from PP, such as information about the last N, to decide whether the PP is attached to VP or NP. LFG also suggests that the reference within two levels is enough. If we restrict the reference path within a few levels, the complexity will not be exponential.

Joshi proposes a class of grammar called \textit{mildly context-sensitive}
grammar (MCG), which is a limited class of unification grammars [Joshi 87]. The set of languages of MCG is called MCL. Its properties are as follows.

- Context-free languages are properly contained in MCL.
- Languages in MCL are parsed in polynomial time.
- MCGs capture only certain kinds of dependency, for example, nested dependencies and limited kinds of crossing dependency\textsuperscript{4}.
- MCL has constant growth property, that is, if the strings of the language are arranged in increasing order of length, then any two consecutive lengths do not differ by arbitrarily large amounts.

2.4.5 Restricting Feature Structures

Features used in unification grammars are categorized into three types:

1. features essential to parse the rest of a sentence
2. features used later to check or disambiguate phrases
3. features representing syntactic and semantic structures of strings but not used to check or disambiguate phrases

The first type includes categories and subcategories. In a sentence with extraposition\textsuperscript{5}, slash categories\textsuperscript{6} will also be included in the first type. Most

\textsuperscript{4}Crossing dependency is found in Dutch but not in English.

\textsuperscript{5}Extraposition is a movement of some phrase to the left of a sentence. An example is the extraposition of wh-NP in the sentence, "Which book did you say that the professor recommended?" In this case, wh-NP is extraposed out of an embedded sentence, "the professor recommended which book."

\textsuperscript{6}Slash categories such as S/NP are used to designate, say, an embedded sentence out of which an NP or a wh-NP has been extraposed.
feature structures are type-3, and this causes the exponential growth of the number of structures. Thus it will be effective to avoid making type-3 until the number of syntactic ambiguities becomes small enough. A simple way is to build only type-1 and type-2 features during parsing, then disambiguate the sentence by using type-2, and finally construct type-3 structures.

But what kind of information is type-2? Obviously type-2 is a subset of type-3. The space and time complexity of parsing depends on how much restriction we can put on type-2 features. One restriction is to limit the feature structures of a node within a few levels of a tree, as discussed in Section 2.4.4. But it is still expensive to use many kinds of feature structure before ambiguities are solved. We thus need to find a set of features that is small enough to avoid exponential explosion of feature structures while informative enough to solve syntactically ambiguous structures. We will propose a solution in Chapter 3.

2.5 Disambiguation

In the case of the sentence:

On one, neurobiologists are trying to chart the hardware of memory, the network of nerve cells — neurons — within the brain whose activity and charges constitute the actual physical basis for memory.

most of the ambiguities are caused by attachments, such as prepositional attachment “within the brain,” and noun reexpressions such as “the hardware of memory, the network of nerve cells.” There are usually many causes of ambiguities, which are multiplied and result in an exponential explosion of
ambiguities. But one observation is that some ambiguities are independent of each other and can be disambiguated separately. Reducing a large problem into smaller problems is one general strategy in problem solving. This is explained by the fact that, when there are three five-way ambiguities, the sum of the number of each choice $5 + 5 + 5$ is much smaller than the total number of readings, $5 \times 5 \times 5$. We will need to represent ambiguities efficiently and solve each ambiguity separately.

Though checking used in DCG has been shown to be inappropriate in Section 2.4.2, the appropriate disambiguation process is not clear yet. Though there have been many studies on attachment preference [Dahlgren 86] [Schubert 86], they have been done separately from those of parsing algorithms. A good paradigm of NLP systems dealing with ambiguities needs to solve these problems. We will propose a solution in Chapter 3.

2.6 Using LR Parsing in NLP

2.6.1 Modifying LR Parsing for NLP

LR parsing is attractive for compiler construction for variety of reasons. It is also attractive in NLP because it uses a top down prediction, which is essential to efficiently parse a sentence with a left-extrapolation, such as wh-question and relative clause. However, it is too much work to construct an LR parser by hand. One needs a specialized tool — an LR parser generator. Fortunately, many such generators are available, one of which is yacc (yet another compiler-compiler). LR parsing table construction is explained in [Aho 86]. LR parsing is described in Appendix A.3.

It is not possible to apply LR parsing to natural languages directly. Since grammars for natural languages are ambiguous, they cannot be parsed
by an LR parser deterministically. Thus, we need to convert a nondeterministic LR parser to an equivalent deterministic one. The new algorithm is called Tomita's algorithm, which is one of the most efficient parsing algorithms for NLP [Tomita 85] [Tomita 86]. It uses dynamic programming.

2.6.2 Conflicts

Every LR parser for a non-LR grammar can reach a configuration in which the parser, knowing the stack and the next input symbol, cannot decide whether to shift or to reduce (a shift/reduce conflict), or cannot decide which of several reductions to make (a reduce/reduce conflict). If there is such a conflict, the action table will have an entry with multi-defined actions. The relationship between these conflicts and attachment preference is explained in [Pereira 85b].

Consider the grammar:

$$
S \rightarrow NP \ VP \\
VP \rightarrow V \ NP \\
VP \rightarrow VP \ PP \\
PP \rightarrow P \ NP \\
NP \rightarrow Det \ N \\
NP \rightarrow NP \ PP
$$

This grammar is the grammar in Appendix A.3 plus the last rule $NP \rightarrow NP \ PP$. This grammar is ambiguous because $PP$ in $VP$ $NP$ $PP$ may be attached to $VP$ or $NP$. Its LR parsing table is shown in Table 2.2. The entries at row 10, column $P$ and at row 11, column $P$ are both multi-defined.

Consider the same input string, $Det \ N \ V \ P \ Det \ N$, as in Appendix A.3. On the input, the parser proceeds along the similar path until step (8). At step (8), the stack is $0 \ NP \ 2 \ V \ 6 \ NP \ 10$, the next input symbol is $P$. Then the action in the configuration, at row 10 column $P$, is multiple-defined. There are two paths that the parser may take from step (8) to step (14) as shown in Table 2.3. The first sequence is similar to that in Table A.2 and
Table 2.2: LR Parsing Table with Conflicts

<table>
<thead>
<tr>
<th>state</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Det N V P $</td>
<td>S NP VP PP</td>
</tr>
<tr>
<td>0</td>
<td>s3 - - - -</td>
<td>1 2</td>
</tr>
<tr>
<td>1</td>
<td>- - - - ac</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>- - s6 s7 -</td>
<td>4 5</td>
</tr>
<tr>
<td>3</td>
<td>- s8 - - -</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>- - - s7 r1</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>- - r6 r6 r6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s3 - - - -</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>s3 - - - -</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>- - r5 r5 r5</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>- - - r3 r3</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>- - r2 s7 r2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r2</td>
</tr>
<tr>
<td>11</td>
<td>- - r4 s7 r4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r4</td>
</tr>
</tbody>
</table>

si: shift state i  
ri: reduce ith production  
ac: accept  
-: error

corresponds to the reading where the PP is attached to the V. The second path corresponds to the reading where the PP is attached to the NP. After step (14a) and step (14b), two paths merge to the same path at step (15).

2.6.3 Tomita’s Algorithm

To handle ambiguous paths of the parser, Tomita introduces a graph-structured stack [Tomita 85] [Tomita 86]. To illustrate how the graph-structured stack works, the two paths of the stack in the previous section is shown in Figure 2.9. At step (8), the graph-structured stack is like the first line in the figure.
0 - NP - 2 V - 6 - NP - 10
\downarrow \quad (8a) \text{ reduce } 2; \text{ goto } 4

0 - NP - 2
\downarrow \quad (9a) (8b) \text{ shift } 7

0 - NP - 2
\downarrow \quad (10a) (9b) \text{ shift } 3

0 - NP - 2
\downarrow \quad (11a) (10b) \text{ shift } 8

0 - NP - 2
\downarrow \quad (12a) (11b) \text{ reduce } 5; \text{ goto } 11

0 - NP - 2
\downarrow \quad (13a) \text{ reduce } 4; \text{ goto } 7; (12b) \text{ reduce } 4; \text{ goto } 5

0 - NP - 2
\downarrow \quad (13b) \text{ reduce } 6; \text{ goto } 10

0 - NP - 2
\downarrow \quad (14a) \text{ reduce } 3; \text{ goto } 4

0 - NP - 2
\downarrow \quad (14b) \text{ reduce } 2; \text{ goto } 4

0 - NP - 2
\downarrow \quad (15) \text{ reduce } 1; \text{ goto } 4

Figure 2.9: Graph-Structured Stack
### Table 2.3: Two Sequences of Stack and Input

<table>
<thead>
<tr>
<th>step</th>
<th>stack</th>
<th>input</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>(8a)</td>
<td>0 NP 2 V 6 NP 10</td>
<td>P Det N $</td>
<td>r2</td>
<td>4</td>
</tr>
<tr>
<td>(9a)</td>
<td>0 NP 2 VP 4</td>
<td>P Det N $</td>
<td>s7</td>
<td></td>
</tr>
<tr>
<td>(10a)</td>
<td>0 NP 2 VP 4 P 7</td>
<td>Det N $</td>
<td>s3</td>
<td></td>
</tr>
<tr>
<td>(11a)</td>
<td>0 NP 2 VP 4 P 7 Det 3</td>
<td>N $</td>
<td>s8</td>
<td></td>
</tr>
<tr>
<td>(12a)</td>
<td>0 NP 2 VP 4 P 7 Det 3 N 8</td>
<td>$</td>
<td>r5</td>
<td>11</td>
</tr>
<tr>
<td>(13a)</td>
<td>0 NP 2 VP 4 P 7 NP 11</td>
<td>$</td>
<td>r4</td>
<td>7</td>
</tr>
<tr>
<td>(14a)</td>
<td>0 NP 2 VP 4 PP 7</td>
<td>$</td>
<td>r3</td>
<td>4</td>
</tr>
</tbody>
</table>

| (8b) | 0 NP 2 V 6 NP 10       | P Det N $ | s7     |      |
| (9b) | 0 NP 2 V 6 NP 10 P 7   | Det N $   | s3     |      |
| (10b)| 0 NP 2 V 6 NP 10 P 7 Det 3 | N $ | s8     |      |
| (11b)| 0 NP 2 V 6 NP 10 P 7 Det 3 N 8 | $   | r5     | 11   |
| (12b)| 0 NP 2 V 6 NP 10 P 7 NP 11 | $       | r4     | 5    |
| (13b)| 0 NP 2 V 6 NP 10 PP 5  | $       | r6     | 10   |
| (14b)| 0 NP 2 V 6 NP 10       | $       | r2     | 4    |

Since multiple states are on top of the stack, the parser needs to perform multiple entries in the table, each of which is also multi-defined. Also, a reduce action may push a new state on top of the stack, and as a result, other actions then become applicable. Thus, in the revised algorithm, the following points must be changed from the algorithm in Figure A.4:

- The shift step is repeated without advancing the pointer to the next input, $p$.
- After all shift actions are completed, $p$ is advanced to the next input symbol, and then begins the reduction step.
- The reduce step is repeated until no more reduction is possible.
- The accept step is checked after the reduce step.
- Error is signaled only when no shift or accept action is applicable for
the current input symbol.

2.7 Toward a Practical System

An effective algorithm useful for parsing long ambiguous sentences will

- be based on tabular parsing algorithms such as the CKY and Earley
- use top down predictions such as LR($k$) parsing algorithm
- restrict the use of feature structures
- use grammar based on CFG linguistic theory
- have an efficient disambiguation routine

An underlining grammar formalism is unification grammars such as LFG.
Chapter 3

Theoretical Basis

In Chapter 2, we stated that we need to restrict the feature structures of complex-feature-based grammars. In this chapter, we propose the head of a phrase as such a feature. First we will explain $\hat{X}$-Theory in linguistics to introduce the dominator, the modifier, and the head of a phrase. Then we will show how the dominator-modifier relation between heads is used to represent ambiguous sentences and to solve ambiguities. We will explore several different alternatives for the dominator-modifier relation. We will also analyze the parsing complexity when the head is used as the feature.

3.1 $\hat{X}$-Theory

The dominator-modifier relation defined in this thesis is based on $\hat{X}$-Theory in linguistics. Chomsky proposed the $\hat{X}$-Convention, also called the $\hat{X}$-Scheme, in "Remarks on Nominalization." $\hat{X}$-Theory was developed in the 1970s and plays an important role in Government-Binding (GB) Theory. It is also adopted by unification grammars, such as GPSG and LFG. $\hat{X}$-Theory is defined as follows [Riemsdijk 86] [Sells 85]:

The head of any phrase is termed $x$, the phrase category containing $x$ is termed $\hat{x}$, and phrase category containing $\hat{x}$ is termed $\bar{x}$.
\[ \bar{N} \rightarrow \text{Determiner } \bar{N} \quad \bar{P} \rightarrow \text{Specifier } \bar{P} \]
\[ \bar{N} \rightarrow N \bar{P} S \quad \bar{P} \rightarrow P \bar{N} \]
\[ \bar{A} \rightarrow \text{Qualifier } \bar{A} \quad S \rightarrow \bar{N} \text{ Auxiliary } \bar{V} \]
\[ \bar{A} \rightarrow A \bar{P} S \quad \bar{V} \rightarrow V \bar{N} \bar{P} S \]

Figure 3.1: Grammar Based on $\bar{X}$-Theory

```
    N
     |  
  Determiner  N  
     |  
     N  P  
     |  
     P  N  
     |  
   a  man on  a hill
```

Figure 3.2: Syntax Structure Based on $\bar{X}$-Theory

For example, NP (Noun Phrase) corresponds to $\bar{N}$. There is an intermediate category $\bar{N}$, which corresponds to NP without a determiner. The head of $\bar{N}$ and $\bar{N}$ is $N$. Using this terminology, the main phrase structure rules of English are written as Figure 3.1, and the tree structure of the phrase "a man on a hill" is shown in Figure 3.2. Since $\bar{X}$ rules and $\bar{X}$ rules look the same for various $X$, it is implied that grammar rules are generalized as in Figure 3.3.

\[ \bar{X} \rightarrow \text{Spec}_X \bar{X} \] for $X = N, V, A, P,$
\[ \bar{X} \rightarrow X \bar{Y} \] and $Y = N, V, A, P$

Figure 3.3: Generalized Grammar Based on $\bar{X}$-Theory
3.2 Head, Dominator, and Modifier

In $\bar{X}$-Theory, a head is defined as categories, such as $N$, $V$, $A$, and $P$ in a main part of English grammar. In this thesis, we make two changes to the definition of a head:

1. The definition of a head is extended to a more general class of context-free grammars.

2. A head is a terminal, instead of nonterminal.

The first change is to allow grammars to cover not only a standard phrase structure rules, but also nonlinguistic structures, such as "the network of nerve cells — neurons — ...." In this example, we use the rule $\bar{N} \rightarrow \bar{N} - N -$. The second change is to accommodate lexical ambiguities as will be explained later.

First, we define a dominator and a modifier. In each grammar rule, one of the categories in the right side is a dominator and others are modifiers. For example, in the first rule of generalized grammar in Figure 3.3, $\bar{X}$ is a dominator and Spec$_{\bar{X}}$ is a modifier. In the second rule, $X$ is the dominator and $\bar{Y}$ is a modifier. We associate the head attribute with each node in a syntactic structure. A head is defined as follows:

- The head of a terminal is the terminal itself.
- For each production\(^1\), the head of a nonterminal of the left side is that of a dominator in the right side.

\(^1\)Production in this context is a synonym of grammar rule.
Figure 3.4: Example Grammar

We say that $m \mathbf{mod} d$ and $d \mathbf{dom} m$ if terminal $m$ is a head of a modifier, and terminal $d$ is that of a dominator. In the definition, it is necessary that no grammar rule has an empty right side. We say that a grammar is $\epsilon$-free if either

- it has no $\epsilon$-production,
- or there is exactly one $\epsilon$-production $S \rightarrow \epsilon$, and then the start symbol $S$ does not appear on the right side of any production.

We use only $\epsilon$-free grammar in this thesis. But any CFG can be converted to an equivalent $\epsilon$-free grammar algorithmically. In $\epsilon$-free grammars, each terminal in a sentence mods another terminal or is a head of the sentence start symbol. For simplicity, we define that the head of a sentence mods a special terminal $[]$. Then in one parse tree, each terminal mods exactly one other terminal.

Consider the grammar in Figure 3.4. Each dominator has the same name as the symbol in the left side of each rule. For example, $V$ is the dominator in $V \overline{N} \overline{P}$. $S$ is commonly used in place of $\overline{V}$. But as we will see later, we must use $\overline{V}$ to describe ambiguities properly. One of the parse trees
Figure 3.5: A Parse Tree and Heads

for the sentence "Time flies like an arrow" is shown in Figure 3.5. Each node is annotated by a head enclosed by (). It resembles a tournament of some game. There are five mod relations for this tree: time mod flies, flies mod [], like mod flies, an mod arrow, and arrow mod flies.

3.3 Ambiguity

Let us now examine an ambiguous sentence. Though there are different kinds of ambiguity, such as scope, semantic, and reference ambiguities, we will discuss only syntactic ambiguities in this thesis. Syntactic ambiguities are classified into two types, each of which is further classified into two.
Figure 3.6: Ambiguous Grammar

1. word usage ambiguities:
   
   (a) lexical ambiguities
   
   (b) ambiguities between general and specific rules

2. structural ambiguities:
   
   (a) modification ambiguities
   
   (b) conjunction ambiguities

Lexical ambiguities are those of word category. For example, word “time” may be a verb, a noun, or an adjective in the sentence “Time flies like an arrow.”

Ambiguity between general and specific rules relates to the function of each word. The following example explains the ambiguity caused by general and specific rules. If the rules in Figure 3.6 are used, then the sentence “It is hard for him to read the book,” has two syntactic analyses shown in Figure 3.7.

By the general rule $\bar{P} \rightarrow P \bar{N}$, “for him” is analyzed as $\bar{P}$. So the sentence is read as a version of “To read the book is hard for him.” By the specific rule $\bar{S} \rightarrow for \bar{N} to \bar{V}$, “for him to read the book” is analyzed as an embedded sentence $\bar{S}$. So the sentence is read as a version of “For him to read the book is hard.” Thus the sentence has two different parse trees.
3.4 mod Relations and Ambiguities

Now we will consider how ambiguities are described in terms of mod relations.

3.4.1 mod Relation and Structural Ambiguities

First, consider the relation between head and the structural ambiguities. There are five readings in the sentence, "I saw a man on a hill with a telescope." Five syntactic trees for $\bar{v}$ are shown in Figure 3.8 and Figure 3.9.

Each reading has ten mod relations, that is, one for each word.
Table 3.1: `mod` Relations in Five Readings

<table>
<thead>
<tr>
<th>mod relation</th>
<th>reading</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>I mod saw</code></td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td><code>saw mod []</code></td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td><code>a mod man</code></td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td><code>man mod saw</code></td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td><code>on mod saw</code></td>
<td>1,4</td>
</tr>
<tr>
<td><code>on mod man</code></td>
<td>2,3,5</td>
</tr>
<tr>
<td><code>a mod hill</code></td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td><code>hill mod on</code></td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td><code>with mod saw</code></td>
<td>1,3</td>
</tr>
<tr>
<td><code>with mod man</code></td>
<td>2</td>
</tr>
<tr>
<td><code>with mod hill</code></td>
<td>4,5</td>
</tr>
<tr>
<td><code>a mod telescope</code></td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td><code>telescope mod with</code></td>
<td>1,2,3,4,5</td>
</tr>
</tbody>
</table>

Eight of them are common for all five readings, but two are different from one reading to the other, as shown in Table 3.1.

A structural ambiguity is represented by a modifier having several different dominators. In this example, `on` may `mod saw` or `man`, and `with` may `mod saw`, `man`, or `hill`. This result suggests that to solve structural ambiguities, we should focus on each word, see it as modifier, and check what dominators it has, rather than focus on the dominator. Some modifications are dependent on each other. For example, `with mod man` and `on mod saw` do not appear in the same reading. This is because we are using CFG, in which two modifications cannot cross each other.
Figure 3.8: Five Syntactic Structures
Figure 3.9: Five Syntactic Structures (Continued)
3.4.2 mod Relations and Word Usage Ambiguities

Second, consider word usage ambiguities. We see that heads of dominators and modifiers are not sufficient to express this kind of ambiguity properly. For example, in the sentence “Time flies like an arrow,” time is a noun and a subject of VP flies, or an adjective modifying N flies. Since we have time mod flies in both cases, the difference is not clearly distinguished by the mod relation. To fix this problem, we need to include some additional information, such as category and functional relation in mod relations. We first consider category and will consider functional relation later.

There are two candidates for categories in mod relations:

1. a preterminal

2. a nonterminal appearing in the rule where a modification occurs

Consider a phrase “a dancing girl,” dancing mod girl. Suppose we use the following productions.

\[ \text{N} \rightarrow \text{Det} \ A \ \bar{N} \]
\[ A \rightarrow V_{\text{ing}} \]
\[ V_{\text{ing}} \rightarrow \text{dancing} \]
\[ V_{\text{ing}} \rightarrow \text{moving} \]

The preterminal of dancing is a present particle \( V_{\text{ing}} \). The first candidate gives the mod relation, \( V_{\text{ing}} \text{ dancing mod N girl} \), while the second gives the mod relation, A dancing mod N girl. But dancing is used as an adjective (A) instead of a verb (V) in this phrase. The second relation describes the actual relation better than the first.
To avoid this inadequacy, we may define *dancing* as an adjective.

\[
\bar{N} \rightarrow \text{Det} \ A \ \bar{N} \\
A \rightarrow \textit{dancing} \\
V_{\text{ing}} \rightarrow \textit{dancing} \\
A \rightarrow \textit{moving} \\
V_{\text{ing}} \rightarrow \textit{moving}
\]

But this makes a grammar larger. Thus we conclude that we should use a nonterminal appearing in a rule as a category.

In this approach, we need to limit grammars in such a way that we will have appropriate \textit{mod} relations. In the grammar in Figure 3.4, we used \( \bar{V} \) in place of \( S \). The reason is now clear: to describe the category of verbs appearing in \textit{mod} relations as \( \bar{V} \). This category will be discussed further in Section 3.4.4.

### 3.4.3 Using Functional Relations in \textit{mod} Relations

Using functional relations in \textit{mod} relations may be useful at the disambiguation step, in part because functional relations are close to semantics. In LFG, a feature structure is called *f-structure*. LFG grammar rules are of the following form:

\[
\bar{V} \rightarrow \bar{N} \quad \bar{V} \\
\quad (\uparrow \text{Subj}) = \downarrow \quad \uparrow = \downarrow \\
\bar{V} \rightarrow V \quad \bar{N} \\
\quad \uparrow = \downarrow \quad (\uparrow \text{Obj}) = \downarrow
\]

The first rule describes that \( \bar{N} \) is the subject feature of \( \bar{V} \), \( \bar{V} \) is the head of \( \bar{V} \), and all the features in \( \bar{V} \) are passed to \( \bar{V} \). The notations:

\[
Y \quad \text{and} \quad Y \\
(\uparrow \text{x}) = \downarrow \quad \uparrow = \downarrow
\]
are called feature annotations. These rules generate an f-structure like:

\[
\begin{array}{c}
\text{Subj} & \text{[ PRED 'I'] } \\
\text{Obj} & \text{[ PRED 'man'] } \\
\text{PRED} & \text{ 'saw < (↑ Subj)(↑ Obj) >'} \\
\end{array}
\]

We associate a semantic relation, such as Subj, Obj, Oblique_{goal} or Adjunct with each phrase. Functional relations make mod relations closer to LFG f-structure. The translation rule is as follows. Features such as Subj and Obj modify PRED in the same level. The head is the functor of PRED, such as I, man or saw. For example, the head of Subj is “I.” The example f-structure is translated into mod relations Subj I mod saw and Obj man mod saw. In general, an f-structure

\[
\begin{array}{c}
\ldots \\
\text{feature}_1 & \text{[ PRED word}_1\text{] } \\
\ldots \\
\text{PRED} & \text{ word}_2 \\
\ldots \\
\end{array}
\]

is translated into a mod relation, feature\textsubscript{1} word\textsubscript{1} mod word\textsubscript{2}.

Using functional relations has an advantage. When we disambiguate whether P attaches to N or V, and no situational information is available, then we check whether each P can be an Oblique of V or not.

An oblique is a PP modifying a verb as an object does, such as a PP “to Mary” modifies a verb “give.” We say that Oblique\textsubscript{goal} is subcategorized by “give.” It is contrasted with a PP “on Monday” modifying “give,” where the PP is called as an adjunct.

Suppose we have a phrase:

\[
\begin{array}{c}
\text{V} & \text{ N } & \text{ P } & \text{ P } \\
\text{saw} & \text{a man} & \text{on a hill} & \text{with a telescope} \\
\end{array}
\]
In this case, "on a hill" is not Oblique of "saw," but "with a telescope" is because "with a telescope" describes a device, and is closely related to "saw."

A simple heuristic for prepositional attachment says:

- If $\bar{P}$ is an oblique of $V$, then $\bar{P}$ is attached to $V$ rather than to $N$.
- Else if $\bar{P}$ is an Oblique of $N$, then $\bar{P}$ is attached to $N$.
- If both are false, then $\bar{P}$ is Adjunct of $V$ or $N$.

In either case, we decide at this point whether $\bar{P}$ is Oblique or Adjunct of $V$ or $N$. To avoid redundant processes in semantic analysis, we need to pass the information whether PP is Oblique or Adjunct to a semantic analyzer.

There is a question whether we use only functional relations or use them with category. Lexical ambiguities are expressed more clearly by categories than with functional relations. For example, with category, a modifier used as a noun is always indicated as $N$, possibly with bars. But by functional relations, it may be indicated as Subj or Obj, thus the category of a word is not clear. Therefore, functional relations should be used as an augmentation to category in mod relations.

Using functional relations may have a disadvantage. Consider the rule:

$$\bar{V} \rightarrow V \bar{P}$$

The $\bar{P}$ may modify $V$ as Adjunct or Oblique. For the phrase "saw a man on a hill with a telescope," we will have two mod relations:

Oblique with mod saw

Adjunct with mod saw
Thus, a mod relation becomes more ambiguous if we attach functional relations. In many cases, the difference between oblique and adjunct is too obscure to solve. Whether we use functional relations in mod relations is a question similar to how much a parser analyzes a sentence and how much it leaves ambiguities to semantic analysis. One solution is to use a set of possible functional relations like:

\{\text{Adjunct, Oblique}\} \text{ with mod saw}

As the ambiguity is solved, inappropriate elements in the set are removed.

Since mod relations using category cannot carry this information, we need some extra mechanism to do so. One way to do so is to attach a functional annotation to each rule. A grammar looks as follows.

\[(A \rightarrow X_1 \ldots X_{i-1} X_i X_{i+1} \ldots X_n), (F_1, \ldots, F_{i-1}, \text{dom}, F_{i+1}, \ldots, F_n)\]

Every \(F_j (j \neq i)\) is a set of functional relations of \(X_j\) to the dominator \(X_i\). Currently, functional relations have not yet been integrated into mod relations in the system implemented in this thesis.

3.4.4 mod Relations and Spurious Ambiguities

We say a grammar is spurious ambiguously if there is more than one parsing tree having the same semantic structure. As a grammar becomes large, spurious ambiguities become a serious problem in parsing. A parser may generate duplicated semantic structures, or generate inconsistent semantic structures.

Consider a noun phrase conjunction. We need the following two new
Figure 3.10: Parse Trees for "men and women"

rules to parse this conjunction.

\[ \overline{N} \rightarrow N \text{ Conj } \overline{N} \]
\[ N \rightarrow \overline{N} \text{ Conj } \overline{N} \]
\[ \text{Conj} \rightarrow \text{and} \]

The first rule is used to parse a phrase "a man and a woman." The second is used to parse a phrase "an old man and woman," where both "man" and "woman" are "old." Then a phrase "men and women" has two syntactic structures as in Figure 3.10. We define "and" as the head of a noun conjunction. mod relations in the two readings are:

- $\overline{N} \text{ men mod Conj and, } N \text{ women mod Conj and}$
- $\overline{N} \text{ men mod Conj and, } \overline{N} \text{ women mod Conj and}$

These two conjunctions describe the same semantics. This implies that bars over $N$ are insignificant. To make two mod relations identical, we omit bars in the mod relations and get $N \text{ men mod Conj and, } N \text{ women mod Conj and}$. Now spurious ambiguities do not affect mod relations.
Table 3.2: Number of \textit{mod} Relations

\begin{tabular}{|l|l|l|c|c|l|}
\hline
\textit{n} & length of sentence & number of readings & \textit{CFG} & number of \textit{mod} relations \\
\hline
1 & 4 & 1 & 7 & 7 & 4 \\
2 & 7 & 2 & 16 & 15 & 8 \\
3 & 10 & 5 & 29 & 25 & 13 \\
4 & 13 & 14 & 47 & 37 & 19 \\
5 & 16 & 42 & 71 & 51 & 26 \\
6 & 19 & 132 & 102 & 67 & 34 \\
7 & 22 & 429 & 141 & 85 & 43 \\
\hline
\end{tabular}

This is necessary to keep the parallelism of f-structure and \textit{mod} relation by the following reason. The relation between \textit{mod} relation and f-structure is one to many. In other words, if there is no ambiguity in the f-structure, there is no ambiguity in the \textit{mod} relation, though the converse is not always true. Moreover, there is only one f-structure even if there is a spurious ambiguity.

\subsection*{3.4.5 Number of \textit{mod} Relations}

We now compare the number of \textit{mod} relations with other measures of ambiguity. The number of \textit{mod} relations calculated for the grammar and sentences in Section 2.4.1 is shown in Table 3.2. When \textit{n} = 1, there are four \textit{mod} relations, one for each word. When \textit{n} changes from \textit{i} – 1 to \textit{i}, the sentence becomes

\textit{I saw a boy (with a girl)}.''}^{i-1}
Then we have another \( i + 2 \mod \) relations:

\[
\begin{align*}
\text{Det} & \quad a & \mod & \ N & \ girl_{i-1} \\
\ N & \ girl_{i-1} & \mod & \ P & \ with \\
\ P & \ with & \mod & \ V & \ saw \\
\ P & \ with & \mod & \ N & \ boy \\
\ P & \ with & \mod & \ N & \ girl_1 \\
\ & \vdots & \vdots & \vdots & \vdots \\
\ P & \ with & \mod & \ N & \ girl_{i-2}
\end{align*}
\]

Therefore, the total number of \( \mod \) relations is

\[
4 + \sum_{i=2}^{n}(i + 2) = 4 + \frac{n(n + 1)}{2} + 2n - (1 + 2) = \frac{n^2 + 5n + 3}{2}
\]

This number is about a half of that of the structures in case of CFG, \( n^2 + 5n + 1 \), and \( \frac{3}{n} \) of the number of rule applications, \( \frac{n^3 + 6n^2 + 20n + 36}{6} \), in Table B.1. This is because \( X_1 w_1 \mod X_2 w_2 \) is not affected by modifiers of \( w_1 \) or those of \( w_2 \). The \( \mod \) relation \( N boy \mod N girl_{i-1} \) represents many cases where NPs having the head \( girl_{i-1} \) with various PPs are attached to \( boy \) with various PPs. Thus \( \mod \) relation is a very compact way to describe ambiguities.

3.4.6 Ambiguities in Terms of \( \mod \) Relations

As stated in Section 3.4.1, we should focus on each word and check its dominators to disambiguate a structural ambiguity. In Section 3.4.2, we said that we need to check each modifier's category to disambiguate a word usage ambiguity. Let us first formally state the relation between ambiguities and \( \mod \) relations.

Suppose that we collect all \( \mod \) relations of all readings of a sentence of length \( n \), \( w_0w_1 \cdots w_{n-1} \). We sort them into \( n \) groups according to
Figure 3.11: Grouping mod Relations of a Sentence

their modifier word, which is the word in the left side of mod, as in Figure 3.11. Let us name these groups \( G_0, G_1, \ldots, \) and \( G_{n-1} \). \( G_i \) is the set of mod relations whose modifier word is \( w_i \). \( m_i \) is the number of mod relations whose modifier word is \( w_i \). Each of \( C_{ij} \) and \( C'_{ij} \) is some category such as \( N, V, A, \) or \( P \).

Now consider each group \( G_i \). Ambiguities are defined as follows:

- If there are some \( j \) and \( k \) such that \( C_{ij} \neq C_{ik} \), the category is ambiguous. Then there is a word usage ambiguity.

- Otherwise if there are some \( j \) and \( k \) such that \( C'_{ij} \neq C'_{ik} \) or \( w'_{ij} \neq w'_{ik} \),
Table 3.3: Example Grouping of mod Relations

<table>
<thead>
<tr>
<th>group</th>
<th>mod relation</th>
<th>reading</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_0$</td>
<td>N time mod V flies</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>V time mod □</td>
<td>2, 3</td>
</tr>
<tr>
<td></td>
<td>A time mod N flies</td>
<td>4</td>
</tr>
<tr>
<td>$G_1$</td>
<td>V flies mod □</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>N flies mod V time</td>
<td>2, 3</td>
</tr>
<tr>
<td></td>
<td>N flies mod V like</td>
<td>4</td>
</tr>
<tr>
<td>$G_2$</td>
<td>P like mod V flies</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>P like mod V time</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>P like mod N flies</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>V like mod □</td>
<td>4</td>
</tr>
<tr>
<td>$G_3$</td>
<td>Det an mod N arrow</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>$G_4$</td>
<td>N arrow mod P like</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td></td>
<td>N arrow mod V like</td>
<td>4</td>
</tr>
</tbody>
</table>

The category is the same but dominators are ambiguous. Then there is a structural ambiguity.

The mod relations of the four readings in Figure 3.12 for the sentence “Time flies like an arrow” with the grammar in Figure 3.4 is shown in Table 3.3. $w_0 = time$, $w_1 = flies$, $w_2 = like$, $w_3 = an$, and $w_4 = arrow$. Consider $G_0$, which corresponds to the word time. Since there are three categories $C_{01} = N$, $C_{02} = V$, and $C_{03} = A$, there is a word usage ambiguity for the word time. Consider $G_2$, which corresponds to the word like. In the first three mod relations in $G_2$, the word like is used as P. But on the right side of these mod relations, three different dominators — V flies, V time, and N flies — appear, indicating a structural ambiguity.
3.4.7 Final Remarks on the mod Relation

As the head has been so far defined, the dominator may be chosen arbitrarily without being based on linguistic theory. For example, we may choose the first nonterminal of the right side of each rule as a dominator. But such a random choice makes the construction of a disambiguation program difficult.

We make a final change on the form of the mod relation. Since the same word may appear more than once in a sentence, we use word position instead of the word itself. Then the final form of the mod relation is \( X_1 \text{ position}_1 \mod \ X_2 \text{ position}_2 \), where \( X_1 \) and \( X_2 \) are categories without bars, such as \( N, V, A, \) and \( P \). \( \text{position}_1 \) and \( \text{position}_2 \) are positions of the words counted from zero.

In the future work, functional relations of modifiers explained in Section 3.4.3 will be added. The mod relation will be

\[
F \ X_1 \text{ position}_1 \mod \ X_2 \text{ position}_2
\]

where \( F = \{ f_1, \ldots, f_n \} \) is a set of possible functional relations.

3.5 General Strategies of Disambiguation

There are many issues in disambiguation. Since ambiguities are dependent on each other, the order of disambiguation affects the efficiency. Possible strategies include the following:

- left-to-right
- top-down
- bottom-up
• disambiguate first where more information is available, then where less is available.

All strategies except top-to-bottom use the fact that in an unambiguous sentence each word modifies only one other word. On the other hand, the top-down strategy can be used when some dominators require obligatory modifiers. For example, a dominator [] requires a modifier, a head of the sentence. Thus, ambiguous modifiers for [] indicate an ambiguity of the verb of the main sentence. Also some verbs require an object. Thus, ambiguous noun modifiers for a verb at the object position indicate an ambiguity of the object of the verb.

Another issue is whether ambiguities should be resolved during parsing or after. There are some tradeoffs in this choice. If some ambiguities are resolved during parsing, the parser will make fewer structures. In other words, the search space is smaller, and therefore the process will be faster. But the parser may fail to find the solution tree later due to a false disambiguation made earlier, so backtracking may be necessary. One alternative to avoid backtracking is to use graph search technique. Another is to resolve only local ambiguities that do not affect the rest of parsing. But solving local ambiguities does not improve the parsing time much because local ambiguities do not affect the parsing time much, as shown in Section 2.4.1. Therefore, in the system explained in Chapter 4, we disambiguate a sentence after parsing.

Though there are many different strategies for disambiguation, mod relations can be used in any strategy. Moreover, mod relations make it possible to disambiguate gradually. Thus, mod relations provide a large flexibility.
Figure 3.12: Four Readings for "Time flies like an arrow."
Chapter 4

Implementation

In this chapter, first the overall structure of the system will be described. Then the data structure and detail of three parts of the system — the compilers, the parser including an example of disambiguation routines, and the debugger — will be set forth.

4.1 System Configuration

The system is implemented by Prolog and consists of three programs: a compiler, a parser, and a debugger. A simplified system configuration is shown in Figure 4.1. There are two ways of parsing a sentence. The parser is intended for parsing a long sentence or many sentences. It includes disambiguation. The debugger is intended for quick correction of a grammar without constructing an LR parsing table. It does not include disambiguation. Each part will be explained briefly in turn.

4.2 Compiler

The compiler consists of four programs: MKGR, MKDIC, LRITEM, and DMG. The relation of programs and data passed between them is shown in Figure 4.2. A grammar consists of two parts: dictionary entries and other grammar rules. The grammar format is a limited form of DCG. It is described below.
Figure 4.1: The System Configuration

Figure 4.2: Compiler System Configuration
MKDIC is a dictionary preprocessor. It takes dictionary entries of root form, such as a verb *fly*, and produces those of inflected form, such as *flies*, *flying* and so on. It also reads a user-defined inflection table that specifies irregular forms of inflected verbs and nouns, such as *am*, *are*, and *is* for the verb *be*. Currently, inflected forms only for verbs and nouns are generated, and those for adjectives are not.

MKGR is a grammar preprocessor. It takes a macro grammar with functional annotations described in Section 3.4.3. To take an advantage of top-down prediction of LR parsing, we must write a grammar with precise subcategorizations, such as the suffix and the arguments of a verb. Since we cannot write a variable in subcategorization due to a limitation of LR parsing, the grammar will be large, having many similar productions. To solve this problem, the system allows users to write a macro grammar. A macro grammar consists of production templates and possible values for each variables in the templates. This greatly reduces the size of the grammar.

LRITEM generates an LR parsing table. DMG is a compiler to integrate a grammar, a dictionary, and an LR parsing table. It produces predicates that are used efficiently by LRPARSER, an LR(0) version of Tomita’s parser. DMG also reads functional annotations and integrates them into the compiled grammar so that the parser will use it to generate mod relations with functional relations\(^1\).

\(^{1}\)This part of DMG has not yet been implemented yet.
4.3 Grammar and Dictionary Format

In this section, a dictionary rule refers to a grammar rule with terminals in the right side. The grammar and dictionary described in this section are translated into three forms of grammar and dictionary, which are used by LRPARSER, CPARSER, and DEBUGGER.

The grammar and dictionary format is basically DCG, but is restricted in the following two ways.

- Prolog programs enclosed by braces ({} can appear only at the end of the right side. This is because in an LR parsing algorithm, the parser can call the program only at a reduce action. The program is optional.
- A Prolog term that represents a nonterminal can have only one or two arguments. It is for the parser to control features and parse a sentence efficiently. The user can specify four kinds of features:
  - subcategory for top-down prediction (type-1 feature)
  - head
  - type-2 features described in Section 2.4.5
  - type-3 features

mod relations are also produced using a category and head of each term in a production.

Grammar and dictionary format is shown in Figure 4.3. The grammar corresponds to the CFG:

\[\text{cat}_0 (\text{subcat}_0) \rightarrow \text{cat}_1 (\text{subcat}_1) \ldots \text{cat}_i (\text{subcat}_i) \ldots \text{cat}_n (\text{subcat}_n)\]

\[\text{cat}_i (\text{subcat}_i) \rightarrow t_1 \ldots t_i \ldots t_m\]
\[ \text{cat}_0(\text{subcat}_0, [\text{Head}_0, \text{feat}_1, \text{feat}_2]) \rightarrow \]
\[ \text{cat}_1(\text{subcat}_1, [\text{Head}_1, \text{feat}_1, \text{feat}_2]), \]
\[ \ldots \]
\[ \text{cat}_n(\text{subcat}_n, [\text{Head}_n, \text{feat}_1, \text{feat}_2]), \]
\[ \{\text{prolog}_\text{program}\}. \]

\[ \text{cat}_0(\text{subcat}_0, [\text{Head}_0, \text{feat}_1, \text{feat}_2]) \rightarrow \]
\[ [t(t_1, \text{Head}_1)], \]
\[ \ldots \]
\[ [t(t_n, \text{Head}_n)], \]
\[ \{\text{prolog}_\text{program}\}. \]

Figure 4.3: Grammar and Dictionary Format

An example grammar is shown in Appendix C.1. Lines beginning with \% are comments. The second line specifies the start symbol \( v([-], \text{main}) \). The rest is the grammar. The corresponding CFG is shown in Appendix C.2.

Examine the terms in productions in the grammar. The first argument of a term, \( \text{subcat}_i \), is a subcategory, that is, a complement of the category and must be ground\(^2\). A term \( \text{cat}_i(\text{subcat}_i) \) is a nonterminal used in parsing. Categories do not include bars, while subcategories do. Thus the nonterminal for the term \( v([-], [-]) \) is \( v([-]) \), meaning \( \vec{v} \). In the example grammar, five categories — n, v, a, p, and end — appear. These categories are used as a head role defined in Section 3.4.2. \( \text{subcat}_i \) is optional. In the first production of the example grammar, end(\_\_) has only one argu-

\(^2\)Containing no variables.
ment: an anonymous variable, underbar "_". Its first argument is considered to be missing, and the second argument to be "_". Thus the nonterminal is simply end without any argument.

The second argument is a feature structure, a list containing two or more elements. In a unary production, the second argument is passed untouched to the left side. The first element is the head, which is a variable Head_j. The same variable Head_i appears in the dominator of the right side. The variable Head_i will be instantiated to an integer, which represents the position of the head of the node in a parse tree. The position of the first word in a sentence is zero. The head of a modifier is "_". The mod relations:

\[ \text{cat}_j \text{ Head}_j \text{ mod } \text{cat}_j \text{ Head}_j (j = 1, \ldots, i - 1, i + 1, \ldots, n) \]

will be generated for the rule.

In the second production of the example grammar:

\[ v([--],[\text{H},\emptyset,\text{subj=N}[V]]) \rightarrow n(--,[_,-,\text{N}]), v([-,\text{tns}],[\text{H},_\text{I}V]). \]

the second term \( v([-,\text{tns}],[\text{H},_\text{I}V]) \) is the dominator, while the first term, \( n(--,[_,-,\text{N}]) \), is a modifier. In one reading of "Time flies like an arrow," H in n(--,[H,_I N]) for "Time" is instantiated to 0, meaning that "Time" is the head. H in v([--],[H,_I V]) for "flies like an arrow" is instantiated to 1, meaning that "flies" is the head. This production will generate a mod relation n time mod v flies.

The second element feat.1, is called feature structure 1 and corresponds to type-2 feature. It can be of any form and will be instantiated by

---

3A production with only one term in its right side.
LRPARSER or DEBUGGER. In the example grammar, the feature structure 1 will be always instantiated to a null list \([\square]\).

The rest of a list, that is \textit{feat\_2}, is called feature structure 2 and corresponds to type-3 feature. It must be a list, and will be instantiated by DEBUGGER or CPARSER. In the same production above, the feature structure 2 of \(n(\_,\_,[N])\) and \(v(\_,\_,\text{tns},[N,\_,V])\) are variables, \(N\) and \(V\), each of which will be instantiated to a list. Suppose that \(N\) is instantiated to \([\text{pred=flies}]\) and \(V\) is instantiated to \([\text{pred=time}]\). The feature structure 2 of a term \(v(\_,\_,[\text{subj=N},V])\), which is \([\text{subj=N},V]\), will be \([\text{subj=}[\text{pred=time}],\text{pred=flies}]\). \textit{feat\_1} and \textit{feat\_2} may contain variables.

In a grammar rule, the value of head is passed from the right side to the left side in bottom-up fashion. We change the definition of terminals so that the variables are instantiated at the terminal level. A terminal is no longer an atom but a pair \(t(\text{terminal},\text{Pos})\), where \text{terminal} is a word, and \text{Pos} is a variable specifying the position of \text{terminal} in the sentence. This variable will be instantiated at the beginning of parsing. Also, an input to a parser is not a list of words, but a list of word and position pairs \([t(\text{word}_1,0),t(\text{word}_2,1),\ldots,t(\text{word}_n,n-1)]\). For example, the input corresponding to a sentence "Time flies like an arrow" will be:

\([t(\text{time},0),t(\text{flies},1),t(\text{like},2),t(\text{an},3),t(\text{arrow},4),t(\text{.'},5)]\).

4.4 Parser

4.4.1 Configuration

The system is intended to be a general purpose syntactic analyzer, possibly integrated with a semantic analyzer or a discourse analyzer. Cur-
Figure 4.4: Parser System Configuration

Currently, the input is a sentence, but extension to multiple-sentences is easy. The parser consists of three programs, LRPARSER, DISAMB, and CPARSER. The programs and data passed between them are shown in Figure 4.4.

LRPARSER is the LR(0) version of Tomita's parser and its user interface. LR(0) parsing will be explained later. It parses the input sentence and generates an ambiguous parse tree and mod relations.

DISAMB is a disambiguation program. It disambiguates a sentence using mod relations and removes inappropriate mod relations and sub-trees. Any knowledge, including syntactic, semantic, and pragmatic information, may be used. The resulting tree may still be ambiguous when enough knowl-
edge is not available. In this thesis, two processes LRPARSER and DISAMB are separate for simplicity. But in a more sophisticated system, they should be interwoven to each other.

CPARSER\textsuperscript{4} is a program that builds a complete feature structure for a sentence described in Section 4.3, which will be the final output of the system.

4.4.2 Output of LRPARSER

Before explaining LRPARSER and its data structures, we need to explain the output data structures. There are five kinds of output:

- one sentence_length/1: the length of the sentence
- one this_sentence/1: the list of each word and its position
- a set of dm/2's: dom relations
- a set of n/4's: nodes in an ambiguous tree
- a set of c/2's: edges in an ambiguous tree

The output for the sentence "Time flies like an arrow" is shown in Appendix C.3.1. There are 16 dm/2's, 31 n/4's, and 32 c/2's.

Each dm/2 represents a dom relation, the inverse of a mod relation. The first argument represents a dominator, and the second represents a modifier. Each consists of two elements, the position and the category of a word. If a modifier is the head of a sentence, the dominator is []. There

\textsuperscript{4}It has not been implemented yet.
are 16 dm/2's in the example. 13 of them are the same as mod relations in Table 3.3. The other three are for the period to modify an ambiguous verb.

Each n/4 represents a node in an ambiguous parse tree. The first argument is the nonterminal of the node. The second and third arguments describe the left and the right boundaries of the node in the sentence. The fourth argument is a list consisting of the head position and feature structure 1 described in Section 4.3. For example, the last three n/4's describe the v([-,-,main]) nodes. Their position is 0 through 6, meaning that their leaf nodes are "Time" through the period. They have different head positions, 2, 1, and 0, which means that the head may be "like," "flies," or "Time." Their feature structure 1 is [].

Each c/2 represents a parent node and its children nodes. The first argument is a parent n/4. The second argument is a list of children n/4's. A set of c/2's represents edges in an ambiguous tree. Though c/2's are enough to describe a tree and n/4's are not necessary, we use n/4's to efficiently identify a node in a tree.

Since a grammar even for a small subset of English is not in a class of LR(0) or LR(1), LRPARSER may assert dm/2's, n/4's, or c/2's that do not contribute to the resulting ambiguous tree. These predicates must be retracted before DISAMB is called. For the sentence "Time flies like an arrow," predicates in Appendix C.3.2 are retracted.

4.4.3 LR(0) Parser

In the system described in this thesis, a simpler LR(0) parser is used in place of an LR(1) parser. An LR(0) parser uses no look ahead symbol. Therefore entries in an action table are the same for all columns. The goto
The LR(0) parsing with dynamic programming works similarly to Earley’s algorithm. The difference is that the LR(0) parsing keeps states while Earley’s algorithm keeps items. As shown in Figure 4.5, items of Earley’s algorithm are equivalent to the union of the sets of items corresponding to each state on top of the stack in the LR(0) parsing. Let us compare Earley’s algorithm in Appendix A.2 with LR parsing algorithm in Appendix A.3. Step (4) in Earley’s algorithm corresponds to the shift step in LR parsing. Step (5) corresponds to the reduce step. However, step (6) is not necessary in LR parsing. Step (6) of Earley’s algorithm is precomputed in the LR(0) parsing.

Both algorithms have the same time and space complexity, $O(n^3)$ and $O(n^2)$. But since several items are represented by one state, constant factors of the LR(0) parsing are much smaller than those of Earley’s algorithm.
4.4.4 An Alternative LR(0) Parsing Table

The system uses an alternative format for LR(0) parsing tables. A goto table consists of a set of goto/3's. An equation of goto function \( \text{goto}[s_1, A] = s_2 \) corresponds to the predicate \( \text{goto}(s_1, A, s_2) \).

Shift action is also represented by goto/3's. For efficiency, a new predicate, from/1, is introduced. A set of arguments of from/1's represents states at which shift actions may take place. from(s) is true if and only if there are some preterminal \( A \) and goto(s, A, ..). from/1's and goto/3's corresponding to the grammar in Appendix C.1 are shown in Appendix C.4. States are named i1 through i22 in place of 0 through 21. The predicate, start_state/1, in the second line describes the state at the bottom of the stack, which is pushed at the beginning of parsing.

Reduce actions are represented by a set of setofitem/2's. The first argument is a state at which a reduce action takes place. The second argument is of the form \( \text{item}(A, \alpha, [], []) \). \( \text{item}(A, \alpha, \beta, []) \) corresponds to an LR(0) item \( A \rightarrow \alpha \cdot \beta \). The fourth argument [] is a set of look ahead symbols. It is unused in LR(0) parsing, but is provided for an extension from LR(0) to LALR or LR(1) parsing. \( A \rightarrow \alpha \) is a rule to be reduced. This predicate represents a subset of the set of items. The set of setofitem/2's for the grammar in Appendix C.1 is shown in Appendix C.4.2. DMG will integrate each grammar rule with the states at which a reduce action takes place and produces a set of gr1/8's, which will be used by LRPARSER. gr1/8's are described later.
4.4.5 Stack

Each entry in a graph-structured stack must have (1) an end state, (2) a start state, (3) an associated feature structure, and (4) pointers to the next entry under and above it. Since the associated feature is ambiguous, storing a feature structure in a stack entry causes multiple stack entries for the same nonterminal.

To solve this problem, we add a triple: (1) nonterminal, (2) left position, and (3) right position to each stack entry in place of a feature structure. Then we use n/4's described in Section 4.4.2, to record ambiguous feature structures. Thus a stack entry consists of five elements.

1. nonterm: nonterminal, which is of the form cat(subcat)
2. left_pos: left position, which is an integer
3. right_pos: right position, which is an integer
4. start_state: start state, which is a state under the nonterminal
5. end_state: end state, which is a state above the nonterminal

A triple (nonterm,left_pos,right_pos) points to all the n(nonterm,left_pos,right_pos,feature) with various feature. A pair (left_pos,start_state) points to connecting stack entries below it. A pair (right_pos,end_state) points connecting stack entries above it. Since the number of stack entries is $O(n^2)$ for the sentence of length $n$, a linear list is not efficient to implement stack entries with. To make the access to stack entries faster, entries with the same right_pos are grouped together.

Data structure of the stack is shown in Figure 4.6. The last entry [0-[[],[1,[]]],[[]]] is to indicate the state at the bottom of the stack, which
Figure 4.6: Structure of Stack

is defined by start_state/1. In this case, it is i1. Unlike the ordinary LR parsing, the same entry may be used many times in various reduce actions. But it is complicated to check whether an entry that has already been used in a reduce action may be used later or not. Thus, for efficiency, no entry is removed from the stack.

To illustrate how the stack works, let us trace the stack for the example in Section 4.4.2. First, preterminals of “time,” n, a, and v([inf]), are looked up in a dictionary. By goto function, their end_states are known to be i4, i2, and i10 respectively. The following three entries are pushed on the stack by shift actions.

1- [0, i4, n, i1]
1- [0, i2, a, i1]
1- [0, i10, v([inf]), i1]

By setofitem/2, reduce action of n(-) → n and v([-inf]) → v([inf]) are defined for the states i4 and i10 respectively. The second reduce action pushes a new state i5, at which another reduce action v([-]) → v([-inf]) is defined. Similarly, n(-) is reduced to n(--). Therefore, the
following four stack entries are merged to the stack by reduce actions.

1-[0,i6,n(-),i1]
1-[0,i12,n(--),i1]
1-[0,i15,v([-,-inf]),i1]
1-[0,i11,v([-]),i1]

The rest of the process continues similarly. At the last step, the entry 6-[5,i18,end,i11] is shifted. The reduce action:

\( v([-,-,main]) \rightarrow v([-]) \) end

is defined for the state i18. The connecting entry 5-[0,i11,v([-]),i1] is found. From these two stack entries, the entry 6-[0,i19,v([-,-,main]),i1] is pushed and the parsing ends. The final stack for the example is shown in Appendix C.5.

4.4.6 Compiled Grammar Format

Grammar and dictionary entries shown in Figure 4.3 are transformed to another form used by LRPARSER. The format is shown in Figure 4.7. Any names starting with upper case are variables. Those starting with lower case are nonvariables. A number after \% is the position of the argument. Both rules are Prolog predicates and are directly executed. An example compiled from a grammar in Appendix C.1 is shown in Appendix C.6.

A grammar rule is a predicate gr1/8, which is the first predicate in Figure 4.7. The rule is used for two purposes: as an entry of an action table and as a grammar rule. It describes that at the state \( state_n \), the reduce action of the first rule in Figure 4.3 takes place.

The information of an entry on top of the stack consists of the first and the second arguments. The grammar rule is called when the entry:
gr1(state_n, % 1
  [Pos_{n-1}, cat_n(subcat_n), state_{n-1}], % 2
Stack, % 3
n(cat_0(subcat_0), Pos_0, Pos_n, [Head_{0}, feat_{1,0}]), % 4
[n(cat_1(subcat_1), Pos_0, Pos_1, [Head_{1}, feat_{1,1}]),
  ...]
  n(cat_n(subcat_n), Pos_{n-1}, Pos_n, [Head_n, feat_{1,n}]), % 5
State_0, % 6
[Head_0, cat_0], % 7
[[Head_1, cat_1], ..., [Head_{n-1}, cat_{n-1}],
  [Head_{n+1}, cat_{n+1}], ..., [Head_n, cat_n]]) :- % 8
cky_stack([
  Pos_{n-1} - [Pos_{n-2}, state_{n-1}, cat_{n-1}(subcat_{n-1}), State_{n-2}],
  ...
  Pos_1 - [Pos_0, State_1, cat_1(subcat_1), State_0]], % 1
Stack), % 2
n(cat_n(subcat_n), Pos_{n-1}, Pos_n, [Head_n, feat_{1,n}]),
...
  n(cat_1(subcat_1), Pos_0, Pos_1, [Head_1, feat_{1,1}]),
prolog_program.

gr1(t_n, % 1
  [t(t_1, Pos_0), ..., t(t_n, Pos_{n-1})], % 2
n(cat_0(subcat_0), Pos_0, Pos_n, [Head_{0}, feat_{1,0}]), % 3
[t(t_1, Pos_0), ..., t(t_n, Pos_{n-1})], % 4
[Head_0, []], % 5
[[Head_1, []], ..., [Head_{n-1}, []],
  [Head_{n+1}, []], ..., [Head_n, []]]) :- % 6
prolog_program,
Pos_n is Pos_0 + n.

Figure 4.7: Compiled Grammar and Dictionary Format
\( Pos_n - [Pos_{n-1}, state_n, cat_n(subcat_n), state_{n-1}] \)

is on top of the stack. The stack is passed to each \texttt{gr1/8} by the variable \texttt{Stack}. It first calls \texttt{cky_stack/2}, whose first argument is a list of stack entries corresponding to the first \( n - 1 \) nonterminals in the right side of the production. \texttt{cky_stack} searches those entries in \texttt{Stack}. Then \( n/4 \)'s corresponding to the \( n \) nonterminals are searched. After calling \( n/4 \)'s, \texttt{Head}_1, \ldots, \texttt{Head}_n, \texttt{feat}_1, \ldots, \texttt{feat}_n will be fully instantiated. \texttt{feat}_1, \ldots, \texttt{feat}_n are all removed from the rules when a production is compiled into \texttt{gr1/8}. Finally, \texttt{prolog_program} is called to check \texttt{feat}_1, \ldots, \texttt{feat}_n.

The fourth and fifth arguments are \( n/4 \)'s corresponding to the left side and the right side of a rule. The sixth argument \texttt{State}_0 is later used to find a new state, \texttt{New_state}, by a \texttt{goto} relation:

\[
goto(\texttt{State}_0, \texttt{cat}_0(subcat_0), \texttt{New_state}).
\]

The seventh argument is a two-element list, which is the head of the dominator. The eighth argument is a list of two-element lists, which are the heads of the modifiers. These heads are used by the parser to assert

\[
dm([\texttt{Head}_i, \texttt{cat}_i], [\texttt{Head}_j, \texttt{cat}_j]). \quad (j = 1, \ldots, i - 1, i + 1, \ldots, n)
\]

The dictionary rule is a predicate \texttt{gr1/6}, which is the second predicate in Figure 4.7. The first two arguments are inputs. The other four are the same as the fourth, fifth, seventh, and eighth arguments of \texttt{gr1/8}. The first argument is used as an index to \texttt{gr1/6}'s for efficiency. The second argument indicates the next word to parse. Since the system allows a user to specify idioms, this argument is instantiated to a list of the next word and all words following it.
4.4.7 Disambiguation Program

A left-to-right strategy is adopted. First, DISAMB puts dm/2's into n groups according to their modifier words as in Figure 3.11. The steps below are applied for each group.

1. **Subgrouping dm/2's.** Divide a set of dm/2's into subgroups according to the categories of the word. If there are two or more subgroups, there is a word usage ambiguity.

2. **Solving word usage ambiguities.** Using some knowledge, decide which categories are inappropriate and remove the corresponding dm/2 subgroups. As a side effect of this removal, other related c/2's, n/4's, and dm/2's may become unused and be removed.

3. **Solving structural ambiguities.** For each remaining subgroup, decide which dm/2's are inappropriate. Remove the corresponding dm/2's. As a side effect of this removal, other related c/2's, n/4's, and dm/2's may become unused and be removed.

The output of “Time flies like an arrow” is shown in Appendix C.3.1. dm/2’s for the word *time* are divided into three subgroups:

\[
\{\text{dm}([1, v], [0, a])\} \\
\{\text{dm}([1, v], [0, n])\} \\
\{\text{dm}([], [0, v])\}
\]

We use the heuristics that *time* is generally used as a noun rather than an adjective or a verb. The following actions follow:

1. \text{dm}([1, v], [0, a]) and \text{dm}([], [0, v]) are removed.

2. c/2's directly related to these dm/2's are removed.
c(n(n(--), 0, 2, [1, []]),
    [n(a, 0, 1, [0, [1(time)]]), n(n(--), 1, 2, [1, [p(L), l(fly)]]))]).
c(n(v([---], main)), 0, 6, [0, []]),
    [n(v([---]), 0, 5, [0, []]), n(end, 5, 6, [5, [1(c', ')]])])].

3. Since no other c/2 has n(n(--), 0, 2, [1, []]) or n(v([---], main)), 0, 6, [0, []]) in its first argument, these n/4's are removed.

4. c/2's having these n/4's in their right side are removed.

Similar removals climb up and down the parse tree. Finally, all dm/2's corresponding to c/2's that have been removed are also removed. As a result, a total of 10 dm/2's, 17 c/2's, and 16 n/4's are removed. The remaining 6 dm/2's, 15 c/2's, and 15 n/4's represent an unambiguous tree corresponding to reading 1 in Figure 3.12. There is exactly one dm/2 for each word as follows:

    dm([1, v], [0, n]).
    dm([4, n], [3, det]).
    dm([2, p], [4, n]).

    dm([1, v], [2, p]).
    dm([1, v], [5, end]).
    dm([], [1, v]).

Let us now consider a structural ambiguity. There are 14 dm/2's, including 13 mod relations in Table 3.1 and a mod relation for the period at the end of the sentence. The dm/2's for the sentence "I saw a man on a hill with a telescope" are divided into eleven groups:

{ {dm([1, v], [0, n])} 
{ {dm([], [1, v])} } 
{ {dm([3, n], [2, det])} } 
{ {dm([1, v], [3, n])} } 
{ {dm([1, v], [4, p]), dm([3, n], [4, p])} } 
{ {dm([6, n], [5, det])} } 
{ {dm([4, p], [6, n])} } 
{ {dm([1, v], [7, p]), dm([6, n], [7, p]), dm([3, n], [7, p])} } 
{ {dm([9, n], [8, det])} } 
{ {dm([7, p], [9, n])} } 
{ {dm([1, v], [10, end])} }
Since each group contains one subgroup there is no word usage ambiguity. Consider the process of disambiguating the attachment of "with a telescope." The group of dm/2's for "with" contains one subgroup:

\[ \{ \text{dm}([1,v],[7,p]), \text{dm}([6,n],[7,p]), \text{dm}([3,n],[7,p]) \} \]

\text{DISAMB} uses the pragmatic information that it is likely that "I" used "a telescope" to choose \text{dm}([1,v],[7,p]), which means "with a telescope" is attached to "saw." Therefore, \text{dm}([6,n],[7,p]) and \text{dm}([3,n],[7,p]) are removed. Other removals similar to the last example follow. As a result, only the trees of reading 1 and 3 in Figure 3.8 are left.

4.5 Debugger

Though LR parsing is efficient, it has some disadvantages for debugging a grammar.

- It takes considerable time to generate an LR parsing table compared to parsing a sentence.
- An LR parser can parse only a whole sentence. It is difficult to parse only part of the sentence or part of the grammar. Moreover, a parsing table may be totally unusable because of a small mistake in a grammar.
- It may be difficult to spot an error in the grammar because of the size of the parsing table and the complicated process of the parser.

To solve these problems, another parser should be provided to debug a grammar more directly. DCG and left-corner parser are two such candidates. The system provides a DCG parser. Though the original grammar is
written in DCG, it cannot be used because of left-recursion in it. We need to eliminate left-recursion from the grammar.

Removing left-recursion takes less time than producing an LR parsing table. It is also possible to parse part of the sentence using a part of a grammar. Though the output grammar is not the same as the original grammar, it is more readable than an LR parsing table. Thus tracing will be easier.

The debugger part consists of two programs: NOLEFT and DCGSUB. The relation between programs and data passed between them is shown in Figure 4.8. NOLEFT is a translator to eliminate left recursion from the grammar and dictionary, and produces DCG without left-recursion. DCGSUB is a user interface to the DCG parser.
4.6 Experimental Results

LRPARSER and DISAMB have been tested for the following sentences from [Newsweek 86].

1. Before there is knowledge, there must be memory.
2. Yet few subjects remain so unknown, so obscured in metaphor and myth.
3. According to the ancient Greeks, life is the act of recollecting knowledge the soul forgot at the moment of its birth in a body.
4. Today the scientific journey to the heart of the black hole proceeds along two paths.
5. On one, neurobiologists are trying to chart the hardware of memory, the network of nerve cells - neurons - within the brain whose activity and charges constitute the actual physical basis for memory.

The size of the ambiguous tree before and after disambiguation is shown in Table 4.1. Each column represents:

- the number of n/4's
- the number of c/2's
- the number of dm/2's
- the number of words appearing in modifier position of two or more dm/2's
- the number of readings

For each sentence, the sentence number and length are shown first. The sentence length includes punctuations. Each row represents the data:
just after parsing

- after removing n/4's, c/2's, and dm/2's that do not contribute the result
- after solving ambiguities by choosing a member of a set of exclusive dm/2's if there is any ambiguity

The first row shows that the length of the first sentence is 10. The second row shows that, after parsing the first sentence, there are 33 n/4's, 33 c/2's, and 10 dm/2's. The third row shows that, after removing unused ones, there remain 28 n/4's, 28 c/2's, and 10 dm/2's. The first sentence is unambiguous because its length and the number of dm/2's are the same.

The second sentence has two readings even though there is no ambiguous dm/2. This is because of a spurious ambiguity in "metaphor and myth," which is similar to the one described in Section 3.4.4.

For the third sentence, there are 393 readings. The seven words: "of" at position 10, "recollecting" at position 11, "knowledge" at position 12, "forget" at position 15, "at" at position 16, "of" at position 19, and "in" at position 22 have two or more dm/2's. Since no PP in this sentence is an oblique of verbs or nouns, each PP is attached to an adjacent verb or NP on its left. After disambiguating PPs, three ambiguities, which DISAMB currently cannot solve, remain:

- "recollecting" may be an adjective modifying the noun "knowledge" or a verb.
- "knowledge" may be the object of the preposition "of" or of the verb "recollecting."
- The relative clause "the soul forgot ..." may modify "act" or "knowledge."
Table 4.1: Experimental Result

<table>
<thead>
<tr>
<th></th>
<th>n/4</th>
<th>c/2</th>
<th>dm/2</th>
<th>words with surplus</th>
<th>dm/2</th>
<th>readings</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>after</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>sentence 1, length = 10</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parsing</td>
<td>33</td>
<td>33</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>removing unused</td>
<td>28</td>
<td>28</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>sentence 2, length = 14</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parsing</td>
<td>86</td>
<td>87</td>
<td>17</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>removing unused</td>
<td>41</td>
<td>42</td>
<td>14</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><strong>sentence 3, length = 26</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parsing</td>
<td>213</td>
<td>265</td>
<td>51</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>removing unused</td>
<td>177</td>
<td>228</td>
<td>49</td>
<td>7</td>
<td>393</td>
<td></td>
</tr>
<tr>
<td>choosing dm([9,n],[10,p])</td>
<td>173</td>
<td>220</td>
<td>48</td>
<td>6</td>
<td>262</td>
<td></td>
</tr>
<tr>
<td>choosing dm([15,v],[16,p])</td>
<td>154</td>
<td>183</td>
<td>43</td>
<td>5</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>choosing dm([18,n],[19,p])</td>
<td>130</td>
<td>144</td>
<td>37</td>
<td>4</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>choosing dm([21,n],[22,p])</td>
<td>101</td>
<td>105</td>
<td>30</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td><strong>sentence 4, length = 16</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parsing</td>
<td>70</td>
<td>71</td>
<td>17</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>removing unused</td>
<td>55</td>
<td>56</td>
<td>17</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>choosing dm([6,n],[7,p])</td>
<td>50</td>
<td>50</td>
<td>16</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>sentence 5, length = 36</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parsing</td>
<td>353</td>
<td>488</td>
<td>82</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>removing unused</td>
<td>250</td>
<td>355</td>
<td>80</td>
<td>11</td>
<td>3030</td>
<td></td>
</tr>
<tr>
<td>choosing dm([4,be],[5,v])</td>
<td>204</td>
<td>262</td>
<td>67</td>
<td>10</td>
<td>715</td>
<td></td>
</tr>
<tr>
<td>choosing dm([9,n],[10,p])</td>
<td>197</td>
<td>243</td>
<td>64</td>
<td>9</td>
<td>432</td>
<td></td>
</tr>
<tr>
<td>choosing dm([9,n],[14,n])</td>
<td>165</td>
<td>195</td>
<td>55</td>
<td>8</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>choosing dm([14,n],[15,p])</td>
<td>157</td>
<td>178</td>
<td>52</td>
<td>7</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>choosing dm([14,n],[19,n])</td>
<td>137</td>
<td>151</td>
<td>46</td>
<td>6</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>choosing dm([14,n],[21,p])</td>
<td>129</td>
<td>137</td>
<td>43</td>
<td>5</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>choosing dm([32,n],[33,p])</td>
<td>107</td>
<td>108</td>
<td>37</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
The fourth sentence has one ambiguity of PP attachment, which is solved as in the third sentence.

In the fifth sentence, "trying" can be analyzed in three ways:

1. a verb in the present progressive form
2. a noun
3. an adjective

When the first analysis is possible, the second and third analyses are less probable. Therefore, DISAMB chooses \( dm([4, be], [5, v]) \) and removes \( dm([4, v], [5, n]) \) and \( dm([4, v], [5, adj]) \). After this disambiguation, the number of readings decreases from 3030 to 715. Since ambiguities are multiplied, the first few disambiguations reduce the number of readings very quickly.
Chapter 5

Conclusions

5.1 Conclusions

In the problem of disambiguation, we have seen that the number of mod relations for a sentence is much smaller than the number of ambiguous tree structures. Most of the structural and word usage ambiguities are clearly expressed by mod relations. Moreover, we are able to construct mod relations that are not affected by spurious ambiguities, though spurious ambiguities still affect the parsing time.

In the problem of the complexity of parsing, we used limited feature structures, primarily the head of a phrase. Using the head as a main feature does not affect the time complexity $O(n^3)$ of tabular methods for CFG's. It is useful enough to reduce the number of ambiguities to few enough that the parser can instantiate the complete feature structures of an ambiguous tree.

We also found out that we need to write a grammar so that the mod relation is consistent and useful. We conjecture that a grammar written in this way will be linguistically plausible.

5.2 Future Work

CParser has not yet been implemented. In addition, we need to include functional relations in mod relations as described in Section 3.4.3.

As is seen in Table 4.1, the longer the sentence is, the more unused
structures the parser generates. Using LALR parsing instead of LR(0) parsing will reduce the number of unused structures, and therefore improve the speed of parsing.

Though we have shown some examples of disambiguation, we have not studied the heuristics that are necessary for practical application of this technique. One candidate is prepositional attachment using common sense described in [Dahlgren 86]. For the more sophisticated disambiguation, we need to use semantic and pragmatic information. We also need to experiment with various of the strategies discussed in Section 3.5.

In some cases, just looking at mod relations is not enough for disambiguation, and we need to look at c/2’s. One of such cases is left extrapolation in relative clauses. In the third sentence of the example in Section 4.6, “the soul forgot at the moment of its birth in a body” is a relative clause. Syntactically, it may modify “act” or “knowledge.” In disambiguation, we must first determine using c/2’s that the object of “forgot” is missing in the relative clause. Then we must examine which of “act” or “knowledge” is preferable as the object of “forgot.”

We have derived the head from X-Theory in linguistics. Since the head of a phrase plays an important role in GPSG and many other unification grammar formalisms, we need to study this relation between the head determine in this thesis and the head in unification grammar formalisms. To clarify this relation will help us to determine the head of each production and construct a grammar appropriate for parsing and disambiguating a large class of English sentences.
Appendix A

Parsing Algorithms

A.1 The CKY Algorithm

The CKY algorithm is also called CYK algorithm. The algorithm works for grammars in Chomsky normal form (CNF). A grammar is said to be in CNF if it is $\epsilon$-free and each non-$\epsilon$-production is of the form $A \rightarrow BC$ or of the form $A \rightarrow a$. Any CFG can be converted into an equivalent CNF grammar.

The CKY algorithm is shown in Figure A.1. $x$ is an input string, and $n$ is the length of $x$. $V_{ij}$ is a set of $A$ such that $A \Rightarrow x_{ij}$, where $x_{ij}$ is a string of length $j$ starting at $i$th word. That is, $A$ includes $i$th, $i + 1$st, $\ldots$, $i + j - 1$st word. $\emptyset$ is an empty set. For full proof and discussion, see [Aho 72][Hopcroft 79]. The table of $V_{ij}$ for the sentence "I saw a boy with a girl" with the CFG in CNF in Figure 2.4 is shown in Figure A.2. The VP in $V_{26}$ represents two VPs. One is made from V in $V_{21}$ and NP in $N_{35}$. The other one is made from VP in $V_{23}$ and PP in $V_{33}$.

A.2 Earley's Algorithm

In this section, $N$ is a set of nonterminals. $\Sigma$ is a set of terminals. $P$ is a set of productions. $S$ is the start symbol. $A$ and $B$ are nonterminals. $\alpha$, $\beta$, and $\gamma$ are $(N \cup \Sigma)^*$. $a$ is a terminal. The input sentence is $a_1 \ldots a_n$.

The algorithm uses a data structure called item, which has the form
begin
for $i := 1$ to $n$ do
  $V_{i1} := \{A \mid A \rightarrow a \text{ is a production}
  \text{ and the } i\text{th symbol of } x \text{ is } a\}$;
for $j := 2$ to $n$ do
  for $i := 1$ to $n - j + 1$ do
    begin
      $V_{ij} := \emptyset$;
      for $k := 1$ to $j - 1$ do
        $V_{ij} := V_{ij} \cup \{A \mid A \rightarrow BC \text{ is a production,}
        B \text{ is in } V_{ik}, \text{ and } C \text{ is in } V_{i+k,j-k}\}$
    end
end

Figure A.1: The CKY Algorithm

\[ i \rightarrow \]
\[
I \quad saw \quad a \quad boy \quad with \quad a \quad girl
\]

\[
\begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & NP & V & Det & N & P & Det & N \\
2 & \emptyset & \emptyset & NP & \emptyset & \emptyset & NP \\
\hline
3 & \emptyset & VP & \emptyset & \emptyset & PP \\
\hline
4 & S & \emptyset & \emptyset & \emptyset \\
\hline
5 & \emptyset & \emptyset & NP \\
\hline
6 & \emptyset & VP \\
\hline
7 & S \\
\end{array}
\]

Figure A.2: Table of $V_{ij}$
\[ A \rightarrow X_1 X_2 \ldots X_k \cdot X_{k+1} \ldots X_m, i \]. \( A \rightarrow X_1 \ldots X_m \) is a production, and 0 ≤ i ≤ n. The dot between \( X_k \) and \( X_{k+1} \) is a meta symbol. The integer \( k \) may be 0 or \( n \), in which case the dot is at the left or the right end. For each integer \( j \) such that 0 ≤ j ≤ n, we will construct a list of items \( I_j \) such that \[ A \rightarrow \alpha \cdot \beta, i \] is in \( I_j \) for 0 ≤ i ≤ j if and only if \( S \Rightarrow \gamma A \beta \) and \( \gamma \Rightarrow a_1 \ldots a_i \) and \( \alpha \Rightarrow a_{i+1} \ldots a_j \). This means that the item \[ A \rightarrow \alpha \cdot \beta, i \] in \( I_j \) assures us that we have parsed up to \( a_j \) in the sentence. The sequence of lists \( I_0, I_1, \ldots, I_n \) is called the parse lists for the input string \( w \). The parse succeeds if and only if there is some item of the form \[ S \rightarrow \alpha \cdot 0 \] in \( I_n \), where \( S \) is the start symbol.

Earley’s parsing algorithm is as follows.

Input CFG \( G = (N, \Sigma, P, S) \) and an input string \( w = a_1 a_2 \ldots a_n \) in \( \Sigma^* \).

Output The parse list \( I_0, I_1, \ldots, I_n \).

Method First, we construct \( I_0 \) as follows.

1. If \( S \rightarrow \alpha \) is a production in \( P \), add \[ S \rightarrow \cdot \alpha, 0 \] to \( I_0 \).

Now perform steps (2) and (3) until no new items can be added to \( I_0 \).

2. If \[ B \rightarrow \gamma \cdot, 0 \] is on \( I_0 \), add \( A \rightarrow \alpha B \cdot \beta, 0 \) for all \( a \rightarrow \alpha \cdot B \beta, 0 \) in \( I_0 \).

3. Suppose that \( A \rightarrow \alpha \cdot B \beta, 0 \) is an item in \( I_0 \). Add to \( I_0 \), for all productions in \( P \) of the form \( B \rightarrow \gamma \), the item \( B \rightarrow \cdot \gamma, 0 \) (provided this item is not already in \( I_0 \)).

We now construct \( I_j \), having constructed \( I_0, I_1, \ldots, I_{j-1} \).

4. For each \( B \rightarrow \alpha \cdot a \beta, i \) in \( I_{j-1} \) and \( a = a_j \), add \( B \rightarrow \alpha a \cdot \beta, i \) to \( I_j \).
Now perform steps (5) and (6) until no new items can be added.

(5) Let \([A \rightarrow \gamma', i]\) be an item in \(I_j\). Examine \(I_i\) for items of the form
\([B \rightarrow \alpha \cdot A\beta', k]\). For each one found, we add \([B \rightarrow \alpha A \cdot \beta', k]\) to
\(I_j\).

(6) Let \([A \rightarrow \alpha \cdot \beta', i]\) such that \(B \in \beta\) be an item in \(I_j\). For all \(B \rightarrow \gamma\)
in \(P\), we add \([B \rightarrow \cdot \gamma, j]\) to \(I_j\).

Note that consideration of an item with a terminal to the right of the dot
yields no new items in steps (2), (3), (5) and (6).

A.3 LR Parsing

We will show the LR parsing. For detailed discussion, see [Aho 86].

The schematic model of an LR(1) parser is shown in Figure A.3. In this
section, we use LR in place of LR(1) otherwise indicated.
The parser consists of an input, an output, a stack, a driver program, and a parsing table that has two parts, action and goto. The driver program is the same for all LR parsers such as SLR (simple LR), LALR, and canonical LR. Only the parsing table changes from one parser to the another. The class of grammars that each parser accepts varies according to the parsing table. To construct an LR parsing table, see [Aho 86]. The input contains the string to be parsed, followed by $\$, a symbol used as an end marker to indicate the end of input string. The program reads character from an input buffer one at a time. The program uses a stack of the form $s_0X_1s_1X_2s_2\ldots X_ms_m$, where $s_m$ is on top. Each $X_i$ is a nonterminal or a terminal and each $s_i$ is a symbol called a state. $X_1, \ldots X_m$ are not necessary, but they appear only to clarify the algorithm. The parsing table consists of two parts, a parsing action function action and a state transition function goto.

The program behaves as follows. It determines $s_m$, the state on top of the stack, and $a_i$, the current input symbol. It then consults $\text{action}[s_m, a_i]$, the parsing action table entry for state $s_m$ and input $a_i$, which can have one of the four values:

1. shift $s$, where $s$ is a state
2. reduce by a grammar production $A \rightarrow \beta$
3. accept, and
4. error.

The function goto describe which new state the parser goes to from an old state via a grammar symbol. The LR parsing algorithm is summarized below.

Algorithm LR(1) parsing algorithm
Set \( p \) to point to the first symbol of \( w \$ \);

repeat forever begin
  let \( s \) be the state on top of the stack and \( a \) the symbol pointed to by \( p \);
  if \( \text{action}[s, a] = \text{shift} \ s' \) then begin
    push \( a \) then \( s' \) on top of the stack;
    advance \( p \) to the next input symbol
  end
  else if \( \text{action}[s, a] = \text{reduce} \ A \rightarrow \beta \) then begin
    pop \( 2 \times |\beta| \) symbols off the stack;
    let \( s' \) be the state now on top of the stack;
    push \( A \) then \( \text{goto}[s', A] \) on top of the stack;
    output the production \( A \rightarrow \beta \)
  end
  else if \( \text{action}[s, a] = \text{accept} \) then return
  else error
end

Figure A.4: LR Parsing Program

Input An input string \( w \) and an LR parsing table with functions \( \text{action} \) and \( \text{goto} \) for a grammar \( G \).

Output If \( w \) is in \( L(G) \), a bottom-up parse for \( w \); Otherwise, an error indication.

Method Initially, the parser has \( s_0 \) on its stack, where \( s_0 \) is the initial state, and \( w \) in the input buffer. The parser then executes the program in Figure A.4.

Table A.1 shows an LR parsing table for the grammar:

\[
\begin{align*}
S & \rightarrow \text{NP} \ VP \\
\text{NP} & \rightarrow P \ NP \\
\text{VP} & \rightarrow V \ NP \\
\text{PP} & \rightarrow P \ PP \\
\end{align*}
\]

The parser accepts strings of the form, \( \text{Det} \ N \ V \ \text{Det} \ N \ (P \ \text{Det} \ N)^n \), but not
Table A.1: LR Parsing Table

<table>
<thead>
<tr>
<th>state</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Det N V P $</td>
<td>S NP VP PP</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>s3 - - - -</td>
<td>1 2</td>
</tr>
<tr>
<td>1</td>
<td>- - - - ac</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>- - s5 - -</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>- s6 - - -</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>- - - s8 r1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s3 - - - -</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>- - r5 r5 r5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>- - - r3 r3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s3 - - - -</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>- - - r2 r2</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>- - - r4 r4</td>
<td></td>
</tr>
</tbody>
</table>

si : shift state i
ri : reduce ith production
ac : accept
-- : error

sequences of words. It is explained in Section 2.6 how to modify the parser to parse actual sentences. On the input string, Det N V Det N P Det N, the sequence of stack and input contents is shown in Table A.2. For example, at step (1) the LR parser is in state 0 with Det the first symbol. The action in row 0 and column Det of the action field of Table A.1 is s3, meaning shift and cover the stack with 3. That is what has happened in step (2): the first token Det and the state symbol 3 have both been pushed onto the stack, and Det has been removed from the input. Then N and 6 are pushed in the similar manner. At step (3), the action in row 6 and column V of the action field is r2, meaning to reduce by the second rule NP → Det N in the grammar. 2× | Det N | = 4 symbols, Det 3 N 6, are popped off the stack. State 0 is
Table A.2: Sequence of Stack and Input

<table>
<thead>
<tr>
<th>step</th>
<th>stack</th>
<th>input</th>
<th>act.</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>0</td>
<td>Det N V Det N P Det N $</td>
<td>s3</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>0 Det 3</td>
<td>N V Det N P Det N $</td>
<td>s6</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>0 Det 3 N 6</td>
<td>V Det N P Det N $</td>
<td>r5</td>
<td>2</td>
</tr>
<tr>
<td>(4)</td>
<td>0 NP 2</td>
<td>V Det N P Det N $</td>
<td>s5</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td>0 NP 2 V 5</td>
<td>Det N P Det N $</td>
<td>s3</td>
<td></td>
</tr>
<tr>
<td>(6)</td>
<td>0 NP 2 V 5 Det 3</td>
<td>N P Det N $</td>
<td>s6</td>
<td></td>
</tr>
<tr>
<td>(7)</td>
<td>0 NP 2 V 5 Det 3 N 6</td>
<td>P Det N $</td>
<td>r5</td>
<td>9</td>
</tr>
<tr>
<td>(8)</td>
<td>0 NP 2 V 5 NP 9</td>
<td>P Det N $</td>
<td>r2</td>
<td>4</td>
</tr>
<tr>
<td>(9)</td>
<td>0 NP 2 VP 4</td>
<td>P Det N $</td>
<td>s8</td>
<td></td>
</tr>
<tr>
<td>(10)</td>
<td>0 NP 2 VP 4 P 8</td>
<td>Det N $</td>
<td>s3</td>
<td></td>
</tr>
<tr>
<td>(11)</td>
<td>0 NP 2 VP 4 P 8 Det 3</td>
<td>N $</td>
<td>s6</td>
<td></td>
</tr>
<tr>
<td>(12)</td>
<td>0 NP 2 VP 4 P 8 Det 3 N 6</td>
<td>$</td>
<td>r5</td>
<td>10</td>
</tr>
<tr>
<td>(13)</td>
<td>0 NP 2 VP 4 P 8 NP 10</td>
<td>$</td>
<td>r4</td>
<td>7</td>
</tr>
<tr>
<td>(14)</td>
<td>0 NP 2 VP 4 PP 7</td>
<td>$</td>
<td>r3</td>
<td>4</td>
</tr>
<tr>
<td>(15)</td>
<td>0 NP 2 VP 4</td>
<td>$</td>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>(16)</td>
<td>0 S 1</td>
<td>$</td>
<td>ac</td>
<td></td>
</tr>
</tbody>
</table>

then exposed. Since the goto of state 0 on NP is 2, NP and 2 are pushed onto the stack. We now have the configuration at step (4). Each of the remaining moves is determined similarly.
Appendix B

Derivation of the Complexity of Parsing

The equation for the number of rule applications and structures built by the CKY algorithm is shown in Table B.1. The number of rule application of $\textbf{NP} \rightarrow \textbf{NP PP}$ is calculated as follows. We can apply the rule for varieties of $\textbf{NP}$ and $\textbf{PP}$. There are three parameters: different starting point of $\textbf{NP}$, different length of $\textbf{NP}$ and different length of $\textbf{PP}$. To see them clearly, let us rewrite $(\text{Det } \textbf{N})(\text{Det } \textbf{N})^{n-1}$ as:

$$(\text{Det } \textbf{N})(\text{Det } \textbf{N})^{j-1}(\text{P Det } \textbf{N})^{k}(\text{P Det } \textbf{N})^{n-i-j-k}.$$  

We apply the rule for all $i \geq 0$, $j \geq 1$, $k \geq 1$, such that $i + j + k \leq n$. Thus the number of rule application is

$$\sum_{i=0}^{n-2} \sum_{j=1}^{n-i-1} \sum_{k=1}^{n-i-j} 1 = \sum_{i=0}^{n-2} \sum_{j=1}^{n-i-1} (n-i-j)$$

$$= \sum_{i=0}^{n-2} \frac{(n-i-1)(n-i)}{2}$$

$$= \frac{(n-1)n(n+1)}{6}.$$  

The number of resulting $\textbf{NP}$ structures is also calculated similarly. We do not distinguish two resulting $\textbf{NP}$s with same $i$ and $j+k$. For all $i \geq 0$, $l \geq 1$, such that $i + l \leq n$,

$$(\text{Det } \textbf{N})(\text{Det } \textbf{N})^{j-1}(\text{P Det } \textbf{N})^{l}(\text{P Det } \textbf{N})^{n-i-l-1}.$$
Table B.1: Equations for CFG

<table>
<thead>
<tr>
<th>Rule</th>
<th>Number of Rule Application</th>
<th>Number of Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow NP \ VP$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>$NP \rightarrow$ Det $N$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>$NP \rightarrow$ NP $PP$</td>
<td>$\frac{n(n-1)}{2}$</td>
<td>$\frac{(n+1)n(n-1)}{6}$</td>
</tr>
<tr>
<td>$PP \rightarrow$ P $NP$</td>
<td>$\frac{n(n-1)}{2}$</td>
<td>$\frac{n(n-1)}{2}$</td>
</tr>
<tr>
<td>$V \rightarrow$ V $NP$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>$VP \rightarrow$ V $NP$</td>
<td>$\frac{n(n-1)}{2}$</td>
<td>$n$</td>
</tr>
<tr>
<td>$VP \rightarrow$ VP $PP$</td>
<td>$\frac{n(n-1)}{2}$</td>
<td>$n$</td>
</tr>
<tr>
<td>Subtotal</td>
<td>$\frac{n^3+6n^2+11n}{6}$</td>
<td>$n^2 + 2n$</td>
</tr>
<tr>
<td>$Det \rightarrow a$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>$NP \rightarrow I$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>$N \rightarrow boy$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>$N \rightarrow girl$</td>
<td>$n - 1$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>$P \rightarrow with$</td>
<td>$n - 1$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>$V \rightarrow saw$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>Total</td>
<td>$\frac{n^3+6n^2+29n+8}{6}$</td>
<td>$n^2 + 5n + 1$</td>
</tr>
</tbody>
</table>
Therefore, the number of resulting NP structures is

\[
\sum_{i=0}^{n-2} \sum_{i=1}^{n-i-1} 1 = \sum_{i=0}^{n-2} (n - i - 1) = \frac{n(n-1)}{2}.
\]

In case of complex-feature-based grammar, we need to consider an internal structure of each nonterminal. For example, NP = (Det N)(P Det N)^i has \(\text{Cat}_i\) different structures due to the variety of PP attachment, where \(\text{Cat}_i\) is the \(i\)th Catalan number. \(\text{Cat}_i\) is defined as

\[
\binom{2i}{i} - \binom{2i}{i-1}.
\]

\(\text{Cat}_n\) represents the number of readings in a phrase that has \(n\) post-modifiers [Church 82]. First several are \(\text{Cat}_0 = 1, \text{Cat}_1 = 1, \text{Cat}_2 = 2, \text{Cat}_3 = 5, \text{Cat}_4 = 14, \ldots\). So the ambiguities of the sentence is \(\text{Cat}_{n+1}\) as in Table 2.1.

In the string (Det N)(P Det N)^n, there are \(n - i\) different substrings NP = (Det N)(P Det N)^i due to variety of starting point. Thus, the total number of NPs is

\[
\sum_{i=0}^{n-1} (n - i) \text{Cat}_i.
\]

By similar calculations, we will get equations in Table B.2. Unlike in CFG, the number of rule applications and that of structures built are the same in this case.

\(\text{Cat}_n\) grows exponentially.

\[
\text{Cat}_n = \binom{2n}{n} - \binom{2n}{n-1} = \frac{(2n)!}{n!n!} - \frac{(2n)!}{(n-1)!(n+1)!} = \frac{(2n)!}{n!(n+1)!}.
\]
Table B.2: Equations for Complex-Feature-Based Grammar

<table>
<thead>
<tr>
<th>category</th>
<th>the number of structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>$\sum_{i=0}^{n-1}(n-i)\text{Cat}_i$</td>
</tr>
<tr>
<td>PP</td>
<td>$\sum_{i=0}^{n-2}(n-i-1)\text{Cat}_i$</td>
</tr>
<tr>
<td>VP</td>
<td>$\sum_{i=1}^{n}\text{Cat}_i$</td>
</tr>
<tr>
<td>S</td>
<td>$\sum_{i=1}^{n}\text{Cat}_i$</td>
</tr>
<tr>
<td>subtotal</td>
<td>$(2n-1)\text{Cat}<em>0 + \sum</em>{i=1}^{n-2}(2n-2i+1)\text{Cat}<em>i + 3\text{Cat}</em>{n-1} + 2\text{Cat}_n$</td>
</tr>
<tr>
<td>Det</td>
<td>$n$</td>
</tr>
<tr>
<td>NP(I)</td>
<td>$1$</td>
</tr>
<tr>
<td>N</td>
<td>$n$</td>
</tr>
<tr>
<td>P</td>
<td>$n-1$</td>
</tr>
<tr>
<td>V</td>
<td>$1$</td>
</tr>
<tr>
<td>total</td>
<td>$(2n-1)\text{Cat}<em>0 + \sum</em>{i=1}^{n-2}(2n-2i+1)\text{Cat}<em>i + 3\text{Cat}</em>{n-1} + 2\text{Cat}_n + 3n + 1$</td>
</tr>
</tbody>
</table>
\[
\frac{2n}{n} \cdot \frac{2n-1}{n-1} \cdots \frac{n+2}{2} \geq 2^{n-1}
\]

Thus, the total number of structures grows exponentially.
Appendix C

Examples of Data Structures

C.1 An Example Grammar in DCG

% start symbol
start(v([-,-,main])).

% An example grammar
v([-,-,main], [H, [], [Z]]) -- v([-], [H, [], [Z]], end(-)).

v([-], [H, [], subj=N[V]]) -- n([-], [_, [], [N]], v([-], [tsn], [H, [], [V]])).

v([-], v) -- v([-], inf), v).

v([-], [tsn], v) -- v([-], [tsn], v).

v([-], inf), v) -- v([-], v).

v([-], [tsn], [H, [], obj=N[V]]) -- v([-], v([tsn], [H, [], [V]]), n([-], [_, [], [N]])).

v([-], [tsn], [H, [], obj=N[V]]) -- v([-], v([inf], [H, [], [V]]), n([-], [_, [], [N]])).

v([-], [tsn], [H, [], pp=P[V]]) -- v([-], v([-], v([tsn], [H, [], [V]]), p([-], [_, [], [P]])).

v([-], [inf], [H, [], pp=P[V]]) -- v([-], v([inf], [H, [], [V]]), p([-], [_, [], [P]])).

p([-], [H, [], obj=N[P]]) -- p([-], v([H, [], [P]]), v([-], n([-], [_, [], [N]])));

n([-], [H, [], det=O[N]]) -- det([-], [_, [], [D]]), n([-], v([-], [H, [], [N]])));

n([-], [H, [], adj=P[A]]) -- a([-], v([-], [A]), v([-], n([-], [H, [], [N]])));

n([-], [H, [], adj=P[A]]) -- n([-], [H, [], [N]]), p([-], [_, [], [P]])).

% An example dictionary
end([A, [1, (',', ')]]) -- [t(',', A)].

det([A, [1(af), pred=af]]) -- [t(af, A)].

det([A, [1(an), pred=an]]) -- [t(an, A)].

n([A, [sing, 1(arrow), pred=arrow]]) -- [t(arrow, A)].

n([A, [sing, 1(boy), pred=boy]]) -- [t(boy, A)].

n([A, [pl, 1(fly), pred=fly]]) -- [t(fly, A)].

v([tsn], [A, [pres, 1(fly), pred=fly]]) -- [t(fly, A)].

n([A, [sing, 1(girl), pred=girl]]) -- [t(girl, A)].

n([A, [sing, 1(hill), pred=hill]]) -- [t(hill, A)].

n([-], [A, [1(i), pred='I']]) -- [t(i, A)].

p([A, [1(like), pred=like]]) -- [t(like, A)].

v([tsn], [A, [1(like), pred=like]]) -- [t(like, A)].
C.2 An Example Grammar in CFG

\[ \begin{align*}
  v([\text{-}, \text{main}]) & \rightarrow v([\text{-}]) \quad \text{end} \\
  v([\text{-}]) & \rightarrow n([\text{-}]) v([\text{-}, \text{tns}]) \\
  v([\text{-}]) & \rightarrow v([\text{-}, \text{inf}]) \\
  v([\text{-}, \text{tns}]) & \rightarrow v([\text{tns}]) \\
  v([\text{-}, \text{inf}]) & \rightarrow v([\text{inf}]) \\
  v([\text{-}, \text{tns}]) & \rightarrow v([\text{tns}]) n([\text{-}]) \\
  v([\text{-}, \text{inf}]) & \rightarrow v([\text{inf}]) n([\text{-}]) \\
  v([\text{-}, \text{tns}]) & \rightarrow v([\text{-}, \text{tns}]) p([\text{-}]) \\
  v([\text{-}, \text{inf}]) & \rightarrow v([\text{-}, \text{inf}]) p([\text{-}]) \\
  p([\text{-}]) & \rightarrow p n([\text{-}]) \\
  n([\text{-}]) & \rightarrow \text{det } n([\text{-}]) \\
  n([\text{-}]) & \rightarrow a n([\text{-}]) \\
  n([\text{-}]) & \rightarrow n([\text{-}]) \\
  n([\text{-}]) & \rightarrow n([\text{-}]) p([\text{-}]) \\
  n([\text{-}]) & \rightarrow \text{end } \\
  \text{det} & \rightarrow a \\
  \text{n} & \rightarrow \text{arrow} \\
  \text{n} & \rightarrow \text{boy} \\
  \text{n} & \rightarrow \text{flies} \\
  \text{v([tns])} & \rightarrow \text{flies} \\
  \text{n([\text{-}])} & \rightarrow \text{girl} \\
  \text{n([\text{-}])} & \rightarrow \text{hill} \\
  \text{n([\text{-}])} & \rightarrow \text{l} \\
  \text{p([\text{-}])} & \rightarrow \text{like} \\
  \text{v([tns])} & \rightarrow \text{like} \\
  \text{n([\text{-}])} & \rightarrow \text{man} \\
  \text{p([\text{-}])} & \rightarrow \text{on} \\
  \text{v([tns])} & \rightarrow \text{saw} \\
  \text{n([\text{-}])} & \rightarrow \text{telescope} \\
  \text{a([\text{-}])} & \rightarrow \text{time} \\
  \text{n([\text{-}])} & \rightarrow \text{time} \\
  \text{v([\text{inf}])} & \rightarrow \text{time} \\
  \text{p([\text{-}])} & \rightarrow \text{with} \\
\end{align*} \]

C.3 Output of the Parser

C.3.1 Net Output of LRPARSER

\[
\begin{align*}
\text{sentence\_length}(6). \\
\text{this\_sentence}([t(\text{time},0), t(\text{flies},1), t(\text{like},2), \\
\hspace{1cm} t(\text{an},3), t(\text{arrow},4), t(\text{'.'},5)]). \\
\text{dm}([1,n],[0,a]). \\
\text{dm}([0,v],[1,n]). \\
\text{dm}([1,v],[0,n]).
\end{align*}
\]
c(n(a,0,1,[0,1(time)])),
  [t(time,0))]).
c(n(n,0,1,[0,[sng,1(time)])),
  [t(time,0))]).
c(n(v([inf]),0,1,[0,[1(time)])),
  [t(time,0))]).
c(n(n(--),0,1,[0,[sng,1(time)])),
  [n(n,0,1,[0,[sng,1(time)]])).
c(n(n(--),0,1,[0,[sng,1(time)])),
  [n(n(--),0,1,[0,[sng,1(time)]])).
c(n(n,1,2,[1,[plr,1(fly)])),
  [t(flies,1)]).
c(n(v([tns]),1,2,[1,[pres3,1(fly)]]),
  [t(flies,1)]).
c(n(n(--),1,2,[1,[plr,1(fly)]]),
  [n(n,1,2,[1,[plr,1(fly)]])]).
c(n(n(--),0,2,[1,[[]]]),
  [n(a,0,1,[0,[1(time)]]),n(n(--),1,2,[1,[plr,1(fly)]])]).
c(n(n(--),1,2,[1,[plr,1(fly)]]),
  [n(n(--),1,2,[1,[plr,1(fly)]])]).
c(n(v([-inf]),0,2,[0,[[]]]),
  [n(v([inf]),0,1,[0,[1(time)]]),n(n(--),1,2,[1,[plr,1(fly)]])]).
c(n(v([-tns]),1,2,[1,[pres3,1(fly)]]),
  [n(v([tns]),1,2,[1,[pres3,1(fly)]])]).
c(n(p,2,3,[2,[1(like)]]),
  [t(like,2)]).
c(n(v([tns]),2,3,[2,[1(like)]]),
  [t(like,2)]).
c(n(det,3,4,[3,[1(an)]]),
  [t(an,3)]).
c(n(n,4,5,[4,[sng,1(arrow)]]),
  [t(arrow,4)]).
c(n(n(--),4,5,[4,[sng,1(arrow)]]),
  [n(n,4,5,[4,[sng,1(arrow)]])]).
c(n(n(--),3,5,[4,[[]]]),
  [n(det,3,4,[3,[1(an)]]),n(n(--),4,5,[4,[sng,1(arrow)]]))].
c(n(p(--),2,5,[2,[[]]]),
  [n(p,2,3,[2,[1(like)]]),n(n(--),3,5,[4,[[]]]))].
c(n(v([-tns]),2,5,[2,[[]]]),
  [n(v([tns]),2,3,[2,[1(like)]]),n(n(--),3,5,[4,[[]]]))].
c(n(v([-]),0,5,[2,[[]]]),
  [n(n(--),0,2,[1,[[]]]),n(v([-tns]),2,5,[2,[[]]]))].
c(n(v([-tns]),1,5,[1,[[]]]),
  [n(v([-tns]),1,2,[1,[pres3,1(fly)]]),n(p(--),2,5,[2,[[]]])).
C.3.2 Retracted Output of LRPARSER

n(\langle-,-,inf\rangle), 0, 1, [0, [1(time)]]).
n(\langle-,-\rangle), 0, 1, [0, \llbracket(time)\rrbracket]).
n(\langle-,-\rangle), 0, 2, [1([])].
n(\langle-,-\rangle), 0, 2, [0, []]).
n(\langle-, tns\rangle), 2, 3, [2, [1(like)]]).
n(\langle-,-\rangle), 0, 3, [2, []]).
n(\langle-,-\rangle), 0, 5, [1([])].
c(n(\langle-,-,inf\rangle), 0, 1, [0, [1(time)]]),
[\langle-,-\rangle], 0, 1, [0, [\llbracket(time)\rrbracket]]).  
[\langle-,-\rangle], 0, 1, [0, [\llbracket(time)\rrbracket]]).  
[\langle-,-\rangle], 0, 1, [0, [\llbracket(time)\rrbracket]]).  
[\langle-,-\rangle], 0, 2, [1([])].
[\langle-,-\rangle], 0, 2, [0, []]).
c(n(\langle-, tns\rangle), 1, 2, [1, [pres3, 1(fly)]]).
c(n(n(--),0,5,[[1,[]]]),n(a,0,1,[[0,1(time)]]),n(n(--),1,5,[[1,[]]]).)

C.4 An Alternative LR(0) Parsing Table

C.4.1 from/1 and goto/3

% by lrtetm
start_state(i1).
from(i1).
from(i10).
from(i11).
from(i12).
from(i13).
from(i16).
from(i2).
from(i22).
from(i3).
from(i5).
from(i6).
from(i7).
from(i8).
goto(i1,a,i2).
goto(i1,det,i3).
goto(i1,n,i4).
goto(i1,n(--),i6).
goto(i1,n(--),i12).
goto(i1,v([-,-inf]),i5).
goto(i1,v([-]),i11).
goto(i1,v([-,-main]),i9).
goto(i1,v([inf]),i10).
goto(i2,n,i4).
goto(i2,n(--),i8).
goto(i3,n,i4).
goto(i3,n(--),i7).
goto(i6,p,i13).
goto(i6,p(--),i14).
goto(i12,v([-,-tns]),i22).
goto(i12,v([tns]),i16).
goto(i5,p,i13).
goto(i5,p(--),i17).
goto(i11,end,i18).
goto(i10,a,i2).
C.4.2 setofitem/2

% by lriter
setofitem(i14, item(n(--), [n], [], [])).
setofitem(i16, item(n(--), [n(--)], [], [])).
setofitem(i15, item(v([--]), [v([-,-inf]), [], []]).
setofitem(i19, item(sprime, [v([--], main)], [], [])).
setofitem(i10, item(v([-inf]), [v([inf]), [], []]).
setofitem(i18, item(n(--), [a, n(--)], [], [])).
setofitem(i17, item(n(--), [det, n(--)], [], [])).
setofitem(i14, item(n(--), [n(--), p(--)], [], [])).
setofitem(i22, item(v([--]), [n(--), v([-,-tans]), [], []]).
setofitem(i16, item(v([-,-tans]), [v([tans]), [], []]).
setofitem(i17, item(v([-inf]), [v([-,-inf]), p(--)], [], [])).
setofitem(i18, item(v([--], main]), [v([--], end], [], [])).
setofitem(i20, item(v([-inf]), [v([inf]), n(--)], [], [])).
setofitem(i19, item(p(--), [p, n(--)], [], [])).
setofitem(i21, item(v([-,-tans]), [v([-,-tans]), p(--)], [], [])).
setofitem(i15, item(v([-,-tans]), [v([tans]), n(--)], [], [])).
C.5 Stack

[5-[5,118, end, i11],
  [0,39,v([--],main),i1]],
[5-[4,14, n, i3],
  [4,17,n(-),i3],
  [3,119,n(--),i13],
  [3,115,n(--),i16],
  [2,122,v([-,tns]),i12],
  [2,121,p(--),i22],
  [2,117,p(--),i8],
  [2,114,p(--),i8],
  [2,114,p(--),i6],
  [1,18,n(-),i2],
  [1,16,n(--),i10],
  [1,122,v([-,tns]),i12],
  [1,120,n(--),i10],
  [0,15,v([-,inf]),i1],
  [0,112,n(--),i1],
  [0,111,v([--]),i1]],
4-[3,13, det, i16],
  [3,13, det, i13]],
3-[2,116,v([tns]),i12],
  [2,122,v([-,tns]),i12],
  [2,113,p,i8],
  [2,113,p,i8],
  [2,113,p,i6],
  [2,113,p,i5],
  [2,113,p,i22],
  [0,111,v([--]),i1]],
2-[1,14, n, i2],
  [1,18,n(-),i2],
  [1,14,n,i10],
  [1,16,n(--),i10],
  [1,120,n(--),i10],
  [1,116,v([tns]),i12],
  [1,122,v([-,tns]),i12],
  [0,15,v([-,inf]),i1],
  [0,112,n(--),i1],
  [0,111,v([--]),i1]],
1-[0,14, n, i1],
  [0,16,n(-),i1],
  [0,12,a,i1],
  [0,112,n(--),i1],
  [0,110,v([inf]),i1],
C.6 Compiled Grammar

C.6.1 gr1/8

g1(i18, [A, end, B], C, n(v([-,-,main]), D, [E, F, []]),
    [n(v([-]), D, A, [F, G]), n(end, A, E, [H, I])],
    J, [F, v], [[H, end]]) :-
    cky_stack([A-[D,B,v([-]),J]], C),
    n(end, A, E, [H, I]),
    n(v([-]), D, A, [F, G]).

gr1(i22, [A, v([-,-,tns]), B], C, n(v([-]), D, E, [F, []]),
    [n(n(--), D, A, [G, H]), n(v([-,-,tns]), A, E, [F, I])],
    J, [F, v], [[G, n]]) :-
    cky_stack([A-[D,B,n(--),J]], C),
    n(v([-,-,tns]), A, E, [F, I]),
    n(n(--), D, A, [G, H]).

gr1(i5, [A, v([-,-,inf]), B], C, n(v([-]), A, D, [E, F]),
    [n(v([-,-,inf]), A, D, [E, F])], B, [E, v, []]) :-
    n(v([-,-,inf]), A, D, [E, F]).

gr1(i16, [A, v([-,-,tns]), B], C, n(v([-,-,tns]), A, D, [E, F]),
    [n(v([-,-,tns]), A, D, [E, F])], B, [E, v, []]) :-
    n(v([-,-,tns]), A, D, [E, F]).

gr1(i21, [A, v([-,-,inf]), B], C, n(v([-,-,inf]), A, D, [E, F]),
    [n(v([-,-,inf]), A, D, [E, F])], B, [E, v, []]) :-
    n(v([-,-,inf]), A, D, [E, F]).

gr1(i15, [A, n(--), B], C, n(v([-,-,tns]), D, E, [F, []]),
    [n(v([-,-,tns]), D, A, [F, G]), n(n(--), A, E, [H, I])],
    J, [F, v], [[H, n]]) :-
    cky_stack([A-[D,B,v([-,-,tns]),J]], C),
    n(n(--), A, E, [H, I]),
    n(v([-,-,tns]), D, A, [F, G]).

gr1(i20, [A, n(--), B], C, n(v([-,-,inf]), D, E, [F, []]),
    [n(v([-,-,inf]), D, A, [F, G]), n(n(--), A, E, [H, I])],
    J, [F, v], [[H, n]]) :-
    cky_stack([A-[D,B,v([-,-,inf]),J]], C),
    n(n(--), A, E, [H, I]),
    n(v([-,-,inf]), D, A, [F, G]).

gr1(i21, [A, p(--), B], C, n(v([-,-,tns]), D, E, [F, []]),
    [n(v([-,-,tns]), D, A, [F, G]), n(p(--), A, E, [H, I])],
    J, [F, v], [[G, n]]) :-
    cky_stack([A-[D,B,v([-,-,tns]),p(--),J]], C),
    n(p(--), A, E, [H, I]),
    n(v([-,-,tns]), D, A, [F, G]).
J, [F,v], [[H,p]] :-
cky_stack([A-[D,B,v([-,-,tns]),J]],C),
n(p(--),A,E,[H,I]),
n(v([-,-,tns]),D,A,[F,G]).

gr1(i17, [A,p(--),B], C,n(v([-,-,inf]),D,E,[F,[]]),
[n(v([-,-,inf]),D,A,[F,G]),n(p(--),A,E,[H,I])],
J,[F,v],[[H,p]]) :-
cky_stack([A-[D,B,v([-,-,inf]),J]],C),
n(p(--),A,E,[H,I]),
n(v([-,-,inf]),D,A,[F,G]).

gr1(i19, [A,n(--),B], C,n(p(--),D,E,[F,[]]),
[n(p,D,A,[F,G]),n(n(--),A,E,[H,I])],
J,[F,p],[[H,n]]) :-
cky_stack([A-[D,B,p,J]],C),
n(n(--),A,E,[H,I]),
n(p,D,A,[F,G]).

gr1(i7, [A,n(--),B], C,n(n(--),D,E,[F,[]]),
[n(det,D,A,[G,H]),n(n(--),A,E,[F,I])],
J,[F,n],[[G,det]]) :-
cky_stack([A-[D,B,det,J]],C),
n(n(--),A,E,[F,I]),
n(det,D,A,[G,H]).

gr1(i8, [A,n(--),B], C,n(n(--),D,E,[F,[]]),
[n(a,D,A,[G,H]),n(n(--),A,E,[F,I])],
J,[F,n],[[G,a]]) :-
cky_stack([A-[D,B,a,J]],C),
n(n(--),A,E,[F,I]),
n(a,D,A,[G,H]).

gr1(i6, [A,n(--),B], C,n(n(--),A,D,[E,F]),
[n(n(--),A,D,[E,F])],B,[E,n],[]]) :-
n(n(--),A,D,[E,F]).

gr1(i14, [A,p(--),B], C,n(n(--),D,E,[F,[]]),
[n(n(--),D,A,[F,G]),n(p(--),A,E,[H,I])],
J,[F,n],[[H,p]]) :-
cky_stack([A-[D,B,n(--),J]],C),
n(p(--),A,E,[H,I]),
n(n(--),D,A,[F,G]).

gr1(i4, [A,n,B], C,n(n(--),A,D,[E,F]),
[n(n,A,D,[E,F])],B,[E,n],[]]) :-
n(n,A,D,[E,F]).
C.6.2 $\text{gr}1/6$

$\text{gr}1(\text{', ')[t(\text{' '}), A] B),$  
$n(\text{end}, A, C, [A, [\text{l(' ')]}, [t(\text{' '}), A]], [A, [], []]) :-  
C is A+1.$

$\text{gr}1(\text{a}, [t(a, A)] B),$  
$n(\text{det}, A, C, [A, [\text{l(a)]}, [t(a, A)], [A, [], []]) :-  
C is A+1.$

$\text{gr}1(\text{an}, [t(\text{an}, A)] B),$  
$n(\text{det}, A, C, [A, [\text{l(an)]}, [t(\text{an}, A)], [A, [], []]) :-  
C is A+1.$

$\text{gr}1(\text{arrow}, [t(\text{arrow}, A)] B),$  
$n(\text{n}, A, C, [A, [\text{sng, l(arrow)]}, [t(\text{arrow}, A)], [A, [], []]) :-  
C is A+1.$

$\text{gr}1(\text{boy}, [t(\text{boy}, A)] B),$  
$n(\text{n}, A, C, [A, [\text{sng, l(boy)]}, [t(\text{boy}, A)], [A, [], []]) :-  
C is A+1.$

$\text{gr}1(\text{flies}, [t(\text{flies}, A)] B),$  
$n(\text{n}, A, C, [A, [\text{plr, l(fly)]}, [t(\text{flies}, A)], [A, [], []]) :-  
C is A+1.$

$\text{gr}1(\text{flies}, [t(\text{flies}, A)] B),$  
$n(\text{v(}, [\text{tns}], A, C, [A, [\text{pres3, l(fly)]},  
[t(\text{flies}, A)], [A, [], []]) :-  
C is A+1.$

$\text{gr}1(\text{girl}, [t(\text{girl}, A)] B),$  
$n(\text{n}, A, C, [A, [\text{sng, l(girl)]}, [t(\text{girl}, A)], [A, [], []]) :-  
C is A+1.$

$\text{gr}1(\text{hill}, [t(\text{hill}, A)] B),$  
$n(\text{n}, A, C, [A, [\text{sng, l(hill)]}, [t(\text{hill}, A)], [A, [], []]) :-  
C is A+1.$

$\text{gr}1(\text{i}, [t(\text{i}, A)] B),$  
$n(\text{n}, [\text{--}], A, C, [A, [\text{l(i), sng}]}, [t(\text{i}, A)], [A, [], []]) :-  
C is A+1.$

$\text{gr}1(\text{like}, [t(\text{like}, A)] B),$  
$n(\text{p}, A, C, [A, [\text{l(like)]}, [t(\text{like}, A)], [A, [], []]) :-  
C is A+1.$

$\text{gr}1(\text{like}, [t(\text{like}, A)] B),$  
$n(\text{v(}, [\text{tns}], A, C, [A, [\text{l(like)]}, [t(\text{like}, A)], [A, [], []]) :-  
C is A+1.$

$\text{gr}1(\text{on}, [t(\text{on}, A)] B),$  
$n(\text{p}, A, C, [A, [\text{l(on)]}, [t(\text{on}, A)], [A, [], []]) :-  
C is A+1.$

$\text{gr}1(\text{saw}, [t(\text{saw}, A)] B),$  
$n(\text{v(}, [\text{tns}], A, C, [A, [\text{past, l(see)]}, [t(\text{saw}, A)], [A, [], []])$,}
\[ t(saw,A), [A, [], []]) :- \]
C is A+1.
\[ gr1(telescope, [t(telescope,A)|B]), \]
\[ n(n,A,C,[A,[sng,l(telescope)]])], \]
\[ [t(telescope,A)], [A, [], []]) :- \]
C is A+1.
\[ gr1(time, [t(time,A)|B]), \]
\[ n(a,A,C,[A,[l(time)]]), [t(time,A)], [A, [], []]) :- \]
C is A+1.
\[ gr1(time, [t(time,A)|B]), \]
\[ n(n,A,C,[A,[sng,l(time)]]), [t(time,A)], [A, [], []]) :- \]
C is A+1.
\[ gr1(time, [t(time,A)|B]), \]
\[ n(v([inf]), A,C,[A,[l(time)]]), [t(time,A)], [A, [], []]) :- \]
C is A+1.
\[ gr1(with, [t(with,A)|B]), \]
\[ n(p,A,C,[A,[l(with)]]), [t(with,A)], [A, [], []]) :- \]
C is A+1.
BIBLIOGRAPHY


The Theory of Parsing, Translation and Compiling Volume 1.


[Church 82] Kenneth Church and Ramesh Patil.

Coping with Syntactic Ambiguity or How to Put the Block in the Box on the Table.


Using Commonsense Knowledge to Disambiguate.


Generalized Phrase Structure Grammar.

[Hopcroft 79] John E. Hopcroft and Jeffrey D. Ullman.

Introduction to Automata Theory, Languages, and Computation.
Addison Welsley, Reading MA, 1979.
[Joshi 87] Aravind K. Joshi.

Unification and Some New Grammatical Formalisms

[Kaplan 83] Ronald Kaplan and Joan Bresnan.


Algorithm Schemata and Data Structures in Syntactic Processing.

[Kay 84] Martin Kay.


BUP: A Bottom-up Parser Embedded in Prolog.


Horn Clause Logic with Parameterized Types for Situation Semantics Programming.

Unification over Complex Indeterminates in Prolog.


[Newsweek 86] .

Memory.

NEWSWEEK, September 29, 1986.


Principles of Artificial Intelligence.


Definite Clause Grammars - A Survey of the Formalism and a Comparison with Augmented Transition Networks.


[Pereira 85a] Fernando C. N. Pereira.

A Structure-Sharing Representation for Unification-Based Grammar Formalisms.


[Pereira 85b] Fernando C. N. Pereira.

A New Characterization of Attachment Preference.


[Pereira 87] Fernando C. N. Pereira.

Information, Unification and Locality.


Ambiguity Procrastination.


[Riemsdijk 86] Henk van Riemsdijk and Edwin Williams.

Theory of Grammar.


Are there preference trade-offs in attachment decisions?


[Sells 85] Peter Sells.


CSLI, Stanford University, Stanford, CA, 1985.

[Shieber 83] Stuart M. Shieber.

Sentence Disambiguation by a Shift-Reduce Parsing Technique.


[Shieber 84] Stuart M. Shieber.

The Design of a Computer Language for Linguistic Information.


[Shieber 85] Stuart M. Shieber.
Using Restriction to Extend Parsing Algorithms for Complex-Feature-Based Formalisms.


An Introduction to Unification-based Approaches to Grammar.

CSLI Lecture Notes Number 4, CSLI Stanford University, Stanford, CA, 1986.


An Efficient Context-free Parsing Algorithm for Natural Languages.


Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems.


A Parser for Portable NL Interfaces Using Graph-Unification-Based Grammars.


Also appeared as

Parsing as Heuristic Graph Search.

VITA

Takanori Tsukada was born in Memanbetsu, Hokkaido, Japan, on December 4, 1959, the son of Fukumi Tsukada and Takeshi Tsukada. After completing his work at Kitami Hokuto High School, Kitami, Hokkaido, Japan in 1978, he entered Hokkaido University, Sapporo, Japan. He received the degree of Bachelor of Science in Physics from Hokkaido University in March, 1982. He has been employed as a software system engineer in Hitachi Software Engineering Co., Ltd. since 1982. In September, 1985, he received a scholarship from Hitachi Software Engineering Co., Ltd. and entered the Graduate School, Department of Computer Sciences, the University of Texas at Austin.

Permanent address: 6-81 Onoe-machi, Naka-ku
Yokohama 231, Japan

This thesis was typeset\cite{1} with LATEX by the author.

\begin{footnote}{
LATEX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's \TeX program for computer typesetting. \TeX is a trademark of the American Mathematical Society. The LATEX macro package for The University of Texas at Austin thesis format was written by Khe-Sing Teh.}
\end{footnote}