A Dependency-Based Parser Evaluation Method

by

Wei Xiao

A thesis
presented to the University of Manitoba
in partial fulfilment of the
requirements for the degree of
Master of Science
in
Computer Science

Winnipeg, Manitoba, Canada, 1997

©Wei Xiao 1997

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-23555-6

Canadä

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES
*****
COPYRIGHT PERMISSION PAGE

A DEPENDENCY-BASED PARSER EVALUATION METHOD

BY

WEI XIAO

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree

of

MASTER OF SCIENCE

Wei Xiao 1997 (c)

Permission has been granted to the Library of The University of Manitoba to lend or sell copies of this thesis/practicum, to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film, and to Dissertations Abstracts International to publish an abstract of this thesis/practicum.

The author reserves other publication rights, and neither this thesis/practicum nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Manitoba requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

As the emergence of broad-coverage parsers, the "automated quantitative parser evaluation" becomes more important to parser researches. Dependency-based parser evaluation method, one of automated quantitative parser evaluation method, is based on dependency syntax theory and provides a better solution to parser evaluation problem than other evaluation schemes.

This thesis explores the area of dependency-based parser evaluation. Several problems are addressed. Among those are evaluation metrics, modification tools, and evaluation for ambiguous sentences.

An experiment of the dependency-based evaluation method is also performed. The result is reported in this thesis.

Acknowledgements

I would like to thank my supervisor, Dr. Dekang Lin for his guidance, advice, and encouragement over the years. I can not acknowledge his influence enough. I would also like to thank Dr. David Scuse and Kevin Russell for their comments and constructive criticism.

I would like to thank my wife Yongmei, for her moral support and encouragement. Finally, I wish thank my parents and sister without whose encouragement I would not have reached this point.

Contents

1 Introduction **1**

1.1 The Organization of the Thesis 3

2 Related Works **5**

2.1 Parsing Algorithms and Broad-Coverage Parsers 5

2.1.1 Parsing Algorithms 6

2.1.2 Broad-Coverage Parser System 7

2.2 Treebank 10

2.3 The Crossing-Bracket Criterion 11

2.3.1 Evaluation Metrics 11

2.3.2 Pre-process 13

2.4 The Exact Match Criterion 15

2.5 The Problems of Constituency-Based Evaluation 16

3 The Dependency Based Evaluation Method **19**

3.1 Dependency Theory 20

3.1.1 Dependency Syntax 21

3.1.2 Dependency versus Constituency 23

3.1.3 Syntax versus Semantics in Dependency Syntax 25

3.2 Dependency-based Evaluation 25

3.2.1 The Representation of Dependency Trees 26

3.2.2 Evaluation Metrics 28

3.2.3 Selective Evaluation 33
3.2.4 Transforming Constituency Trees into Dependency Trees . . 34
3.2.5 Advantages of the Dependency-Based Evaluation 40

4 A Modification Tool for Dependency Trees 42
4.1 The Objective 42
4.2 Rule 44
4.2.1 Pattern 44
4.2.2 Action 46

5 Implementation and Experimental Results 55
5.1 Class DependencyTree and EvaluationMetrics 56
5.2 Tree Modifying Algorithm 57
5.2.1 Build initial matches 60
5.2.2 Apply actions 62
5.3 Class ModificationModule 66
5.4 Experimental Results 68

6 Evaluation for Ambiguous Sentences 71
6.1 Representation of Multiple Parse Trees 71
6.2 Evaluation Metrics 73
6.3 The Problem of Modifying Parse Forests 75

7 Conclusion 77
7.1 Summary 77
7.2 Future Work 78

A Sample parse trees from treebanks 79

B OMT notations 82

List of Tables

3.1 Dependency relations of some constructions. M indicates modifier and H indicates head. 23

3.2 The dependency structure of the sentence: "The computer will send the files to a printer" 28

3.3 The dependency relations of the sentence: "The computer will send the files to a printer". 38

5.1 The experimental results 69

6.1 Hamming Distance, Recall and Precision 74

List of Figures

2.1 An example of crossing-brackets 12

2.2 recall-precision 14

2.3 An example of misguided scores 16

2.4 A sample analysis of the evaluation metrics 17

2.5 A sample derivation tree of XTAG and its phrase tree 18

3.1 The dependency versus constituency 21

3.2 The semantic structure (a), dependency syntactic structure (b) and constituency syntactic structure (c) of the sentence: "John broke the window with a stone". 26

3.3 The answer and key for the sentence: "The computer will send the files to a printer". 32

3.4 A subset of Magerman's Head Table. 35

3.5 An example for the Head Assignment algorithm 36

3.6 A example of a Relation Table 37

3.7 The structure of XP 38

3.8 The X-bar tree and derived dependency relations of the sentence "The computer will send the file to a printer". 39

3.9 The dependency structures of [the legal rights] and [the [legal rights]] 40

4.1 A pattern and its match 47

4.2 Using rules to transform (a) to (b) and vice verse. The structures circled by dotted line are the matches of the patterns of the corresponding rules. 50

4.3 Transform different analyses of coordinate structures. Tree A and Tree B are two ellided dependency tree for the sentence: "He stood up and gave me letter". 51

4.4 One example of the dependency tree modification 52

4.5 Normalization of pre-infinival 'to' 53

5.1 The architecture of the dependency tree 56

5.2 The Object Model which uses Rumbaugh OMT notation (see Appendix B) 58

5.3 The dependency tree of the sentence: "The computer will send the files to a printer" and a rule pattern. D1, D2, ... are used to identify the nodes of trees. 60

5.4 The example of the Tree Modifying Algorithm 64

5.5 The Object Model which uses Rumgaugh OMT notation 67

6.1 Two possible parses for "flying planes could be dangerous" 72

6.2 An example for modifying a parse forest. The subtrees match the condition are represented by dotted lines 76

B.1 Associations in OMT 82

B.2 Composition (aggregation) in OMT 83

B.3 Classification in OMT 83

Chapter 1

Introduction

The task of an automatic quantitative parser evaluation is to provide a quantitative measure of the accuracy of parsers. This information can be derived by applying parsers to a large number of natural language sentences. With the emergence of broad-coverage parsers, researchers have shown an increasing interest in parser evaluation. Parser evaluation would provide a standard for:

- The comparison of different parser systems. Since parsers are critical components in natural language processing systems, determining the differences between parsers, and which parser should be considered for use are important in solving natural language problems.

- The improvement of a particular system. A quantitative evaluation is also important in the further development of a particular parser. Since a parser is a complex system, it is hard to determine how a specific change to a component of the parser affects the whole system. Evaluation based on a large test data set can help parser developers to estimate if the change harms, or improves the general performance of the parser. In addition, the information derived

from the evaluation allows the researchers to focus on the areas that have the greatest impact on parser performance.

The problem of evaluating parsers can be divided into two subproblems: establishing standard parses and comparing parser generated parses against this standard. Generally parser evaluators use the hand-analyzed parses, or **treebanks**, as the standard to judge parser generated parses. However, the method of the comparison is still open for discussion.

Two types of comparison schemes have been proposed: the constituency-based evaluation (Black [1] and Magerman [17]) and the dependency-based evaluation (Lin [12]). The main difference between them is the syntactic representations of their parses. The former adopts constituent trees to represent both treebanks and parser generated parses, and the latter uses dependency trees. Constituency-based evaluation methods score parsers by comparing constituent boundaries found in treebank parses and parser generated parses; in contrast, the dependency-based evaluation method compares the dependency relations of parses.

The dependency-based method provides a better solution to the parser evaluation problem than the constituency-based method. Two main merits of dependency-based evaluation are:

- The purpose of parsing is usually to facilitate semantic interpretation. Since the semantic structure is embedded in the dependency syntactic structure, the scores of the dependency evaluation are more relevant to how useful a parse is than those of the constituency-based evaluation.

- There are many acceptable ways to analyze some syntactic structures in different parsers. One of the difficult issues in parser evaluation is how to treat these different analyses without bias. A modification tool, which is easy to apply in the dependency-based evaluation method, can modify dependency structures before evaluation to minimize the differences.

This work is based on Lin's evaluation scheme [12]. In the thesis, the earlier dependency-based evaluation scheme is extended in various aspects:

- Besides the Hamming Distance-based measure used in [12], the metric "recall-precision" is introduced to describe two aspects of the system performance: completeness and accuracy.
- The modifying operation in the old evaluation scheme is extended to modify multi-layer dependency tree.
- The problem of the evaluation for ambiguous sentences is also examined.

Furthermore, an experiment is performed to demonstrate the feasibility of the scheme.

1.1 The Organization of the Thesis

Chapter 2 reviews modern broad-coverage parser systems and constituency-based evaluation criteria which are commonly used in parser evaluations.

Chapter 3 examines the dependency-evaluation method by describing the representation of parses and the metrics of evaluation.

The modification tool for the evaluation is presented in Chapter 4. Chapter 5 describes the implementation of the system, followed by the experimental results.

Chapter 6 discusses some issues of the evaluation of ambiguous sentences, and Chapter 7 summarizes the results of the thesis, and suggests possible future areas of study.

Chapter 2

Related Works

The first section of this chapter gives a simple survey of parsing algorithms and the state-of-art broad-coverage parser systems. The second section outlines the concept of the treebank which is used to judge parser generated parses in automatic evaluation. The rest of the chapter reviews the early work on automatic quantitative evaluation methods. Two constituency-based automatic evaluation methods have appeared in the literature so far: the Crossing-Bracket Criterion and the Exact Match Criterion. The main idea of these evaluation criteria is to evaluate parsers by comparing the constituent boundaries identified by the parsers to those implied in the treebanks.

2.1 Parsing Algorithms and Broad-Coverage Parsers

The task of parsing is to determine if a sentence is syntactically well formed and, if so, to find one or more structures for the sentence [20]. Many natural language understanding systems rely on a parser as the first step in processing an input sentence.

There are a number of parsers to have been developed and those parsers are based on different linguistic formalisms and parsing algorithms.

2.1.1 Parsing Algorithms

Parsing algorithm is the computational "device" which encodes the linguistic knowledge to parse natural language sentences [17]. The following are a few examples of the most popular parsing algorithms used among broad-coverage parsers.

Chart parsing algorithm was first presented by Cocke, Kasami and Younger (1967). The main idea of the algorithm is to store intermediate results of parsing in a chart or matrix to cope with redundancy in the parsing search space [6]. A chart enables a parser to keep a record of structures it has already found and information about goals it has adopted. The storage of intermediate results is a time versus space trade-off and turns out to be a key to efficient parsing.

A message passing algorithm was proposed by Lin [16]. The algorithm uses a network to encode grammar. The nodes in grammar network represent grammatical categories and links represent the structural relationships between grammatical categories. An input string is parsed by passing messages in the grammar network. The message passing algorithm is similar to chart parsing; but the function of chart is distributed over the nodes in the network.

Another popular parsing algorithm is Augmented Transition Network (ATN) [23]. Unlike message passing network, ATN can be considered as a finite-state automata

which is augmented with register variable and functional constrains. The parser examines the words of an input string from the left and start to transit over arcs from initial state. The string is grammatical if the final state can be reached.

2.1.2 Broad-Coverage Parser System

Based on early work on natural language parsing, researchers recently expand their efforts to employ new grammatical theories and parsing technologies to build broad coverage parsers for general language.

PRINCIPAR

One example of broad-coverage parsers is the PRINCIPAR system [14, 15]. The PRINCIPAR is a principle-based parser, which makes use of Government-Binding Theory. While rule-based grammars use a large number of rules to describe patterns in a language, GB theory describes these patterns by using more fundamental and general principles. The PRINCIPAR states the GB principles in terms of linguistic concept such as barrier, government and movement, which are relationships between nodes in syntactic structures.

In the PRINCIPAR, the GB principles are directly applied to the description of structures. A structure for the input sentences is only constructed after its description has been found to satisfy all the principles.

The parser is implemented by a message passing algorithm. The grammar is encoded in a network. The nodes in the network are computing agents. They com-

municate each other other by passing messages through the links in the network. The principles are implemented as a set of constrains that must be satisfied during the propagation and process of messages. The constrains are attached to nodes and links in the network.

Statistical techniques in parsing

Many broad-coverage parsers have begun involving statistical technology to solve parsing problem. IBM statistical parser is a well-known example (Black [2]). The grammar of the parser is a feature-based probabilistic context-free grammar (P-CFG). In a P-CFG, probabilities are assigned to each production in the grammar, where the probability assigned to a production, $X \rightarrow Y_1...Y_n$, represents the probability that the non-terminal X is rewritten as $Y_1...Yn$ in the parse of a sentence.

The probabilities can be assigned automatically by using a large manually parsed corpse (treebank) to train the grammar. The statistical task of the parser is to probabilistically train the grammar in order that the parse selected as the most likely one by the parser is a correct parse. The task of the parser is to find the most likely parser in terms of CFG. It is suggested that the use of a large treebank allows the development of sophisticated statistical models that should outperform the traditional approach of using human intuition to develop parse preference strategies [2].

Another broad-coverage parser XTAG [5] also combine statistics strategy with rule-based grammar. XTAG is based on the Tree Adjoining Grammar Formalism (LTAG). LTAG is a lexicalized mildly-context sensitive tree rewriting system

that is related to dependency grammars and categorical grammars [11].

The grammar is encoded by trees in a Tree Database. The parsing of a sentences includes two steps. In the tree-selection step, the parser selects a set of elementary trees from the Tree Database for each lexical item in the sentence. In the tree-grafting step, the selected trees are composed by substitution and adjunct operations.

The XTAG generates parse trees which are ranked by combination of heuristic, which are expressed as structural preference for the derivation of parse trees. In addition, the statistical information about usage frequency of the trees is used to improve the performance of the parser. This information is collected by parsing the Wall Street Journal, the IBM manual, and the ATIS corpus. XTAG consists of a statistics database which contains frequencies of each tree in the Tree Database. In the time of parsing, the parser will first pick the most frequently used trees in the Tree Database.

Using dependency in parsing

One trend in the development of broad coverage parsers is to make use of dependency. The X-bar structure of the PRINCIPAR requires that all phrases must have a head, which is used in almost the same way as in dependency theories. The XTAG uses LTAG formalism and generates derivation trees which capture the dependency between words.

Collins also developed a statistical parser, in which standard probability estimation

techniques are extended to calculate probabilities of dependencies [4]. In Collins' parser, dependencies between pair of words are assigned probabilities and each parse tree can be mapped to a dependency tree. The probability of a parse tree is calculated in terms of the probabilities of the dependency relations in the corresponding dependency tree.

2.2 Treebank

The problem of parser evaluation is to determine whether or not a parser generated parse is correct, and if not, how accurate it is. Early parser evaluation was performed by human evaluators who examined the parses of test sentences and reported the accuracy rate of a parser. However, human judgment is inconsistent, and not a very reliable measure. This means that even if the same parse is evaluated twice by the same evaluator, the results may not be identical; similarly if the same parser is evaluated by different evaluators with different standards and judgments, the results will not be identical.

In automatic evaluation methods, human evaluators are replaced by a treebank. A treebank is a sizable corpus of sentences which have been manually, or semi-automatically parsed. By definition, a treebank parse for any given sentence is considered to be the "correct parse", and is used to judge a parser generated parse. There are some well-known treebanks, such as UPenn [22], Lancaster [2], and Susanne [21]; which are widely used in parser evaluation, statistical parser testing, and training. Appendix A shows some sample parses of those treebanks.

Of course, a treebank may also be internally inconsistent, because it is produced by hand. Unlike human evaluators, the treebankers' standards and judgments are available for review. It is possible to control the quality of the treebank, and to increase its consistency rate to an acceptable level. One way to improve consistency is to have multiple analysts to annotate the same data. In [17], the consistency rate of the treebank was raised from 50% to 90% by applying this method.

2.3 The Crossing-Bracket Criterion

The Crossing-Bracket Criterion was first proposed at DARPA Speech and Natural Language Workshop to rank participating parsers. It has also been used by other researchers to measure the performance of their parsers.

This criterion compares only the constituent boundaries of parser generated parses and treebank parses, and ignores their labels and part-of-speech tags.

2.3.1 Evaluation Metrics

The Crossing-Bracket Criterion consists of two sets of metrics. The first set of metrics includes only one measure: the number of crossing-brackets violations which is defined as follows:

The span of a constituent is defined as the string of words which it dominates, denoted by a pair of indices (i, j) where i is the index of the leftmost word in the constituent, and j is the index of the rightmost word. A single crossing-bracket violation is constituency A with the span (i, j) in a parser generated tree, if there is

constituency B with the span (i', j') in its treebank and $i < i' < j < j'$.

For example, in Figure 2.1, the parse to be evaluated has two crossing-brackets.

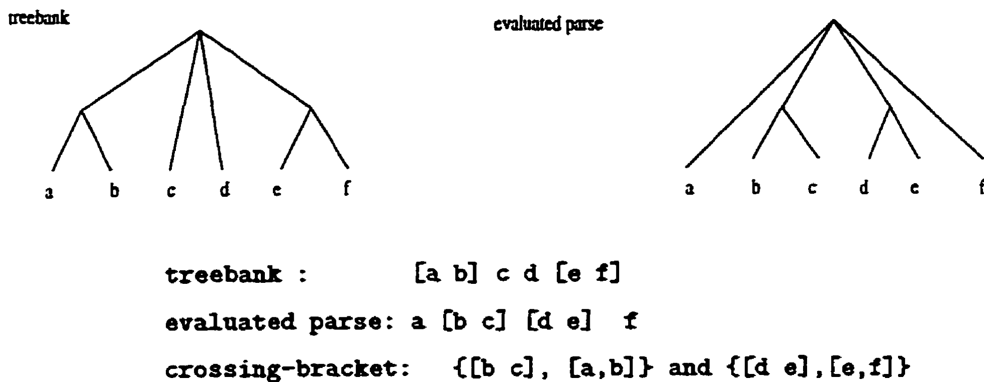


Figure 2.1: An example of crossing-brackets

According to Magerman [17], the crossing-bracket violation, itself, is a weak measure. Consider the example:

```
treebank:        [[All Dallas members] [voted [with Roberts]]
evaluated parse: [All Dallas members voted with Roberts]
the number of crossing-bracket violations: 0
```

The evaluated parse is a very poor parse but yields a perfect score. Thus the number of crossing-bracket violations has to be combined with the second set of the metrics **recall and precision** in order to provide an adequate measure of parser performance.

The concepts of recall and precision were adapted from the field of Information Retrieval. In the Crossing-Bracket Criterion, all constituents in a parser generated parse and its corresponding treebank parse can be classified as three categories:

- the correct constituent: any constituent in both the treebank tree and the parser generated tree
- the possible constituent: any constituent in the treebank tree
- the actual constituent: any constituent in the parser generated parse tree

For a parser generated parse, recall and precision can be computed in terms of correct, possible, and actual constituents:

$$precision = \frac{the\ total\ number\ of\ correct\ constituents}{the\ total\ number\ of\ actual\ constituents}$$

$$recall = \frac{the\ total\ number\ of correct\ constituents}{the\ total\ number\ of\ possible\ constituents}$$

Figure 2.2 shows a sample of the recall-precision analysis.

Precision is the percentage of the constituents in the parser generated parses which match the constituents in the corresponding treebank parses, and recall is the percentage of the constituents in the treebank parses which match the constituents in the parser generated parses.

Recall and precision characterize the different aspects of the performance of parsers. Recall addresses the completeness of parsers and the precision addresses the accuracy. While recall increases, precision tends to decrease and vice versa.

2.3.2 Pre-process

Before the evaluation of a parser, the Crossing-Bracket Criterion erases from input parses all instances of: auxiliaries, 'not', pre-infinitival 'to', null categories, posses-

```
treebank:
[[The odds] [favor [[a special session] [[more [than likely]] [early [in [the ye
ar]]]]]]]
parse:
[[The odds] [favor [a [special session] [[[more than] likely] [early [in the yea
r]]]]]]

treebank only:
  [a special session]
  [than likely]
  [the year]
parse only:
  [special session]
  [more than]

Recall = 8/11 = 72.727272727272734%
Precision  = 8/10 = 80.0%
```

Figure 2.2: recall-precision

sive ending('s and '), and all word-external punctuation marks.

The elements such as auxiliaries, 'not' and 'to' can be analyzed in many different ways in different syntax theories. The erasure of these elements allows the evaluation criterion no bias towards different theories.

2.4 The Exact Match Criterion

In [17], Magerman presented the Exact Match Criterion to evaluate his SPATTER statistical parser. The measure of this criterion is the percentage of the sentences which are correctly parsed. A parse tree is considered to be correct if and only if every constituent, constituent label, and part of speech tag in the parse tree matches those in the treebank analysis.

There are strong arguments against the Exact Match Criterion. It is difficult to reach a consensus about a constituent label or part-of-speech tag set between different parsers. In this method, a single error, as well as multiple serious errors, are treated alike, since all errors in the parse are counted only once. The degree of the correctness of a parse should be taken into account, since it decides how useful the parse is.

2.5 The Problems of Constituency-Based Evaluation

In the Crossing-Bracket Criterion, it is not necessarily true that a parse tree, which has higher evaluation scores, is closer to the correct parse than other parse trees. A very poor parse may be assigned a high score. In [12], Lin gave a few examples in which the method produces misguided scores (see Figure 2.3). It is obvious that *parse a* has a lot more in common with the treebank than *parse b*; however, *parse a* has much lower scores than *parse b* in terms of precision, recall and the number of crossing-brackets.

```
treebank:
  [I [saw [[a man] [with [[a dog] and [a cat]]]] [in [the park]]]]

parse a:
  [I [saw [[a man] [with [[a dog] and [a cat] [in [the park]]]]]]]

  # of crossing brackets=3; recall=60%; precision=63.6%

parse b:
  [I [saw [a man] with [a dog] and [a cat] [in [the park]]]]

  # of crossing brackets=0; recall=100%; precision=70%
```

Figure 2.3: An example of misguided scores

Another problem is that the method is too sensitive to the granularity of a parse. Treebank parses are constructed as "skeleton parses" because not all constituents

will always figure in a treebank parse. Therefore, the parse generated by a parser is usually more detailed in its representation than a treebank parse. For example, in the Lancaster treebank, some internal noun-phrase structures are considered to be nonessential, and are omitted by treebankers. The UPenn treebank parses are even more shallow than the Lancaster treebank. The UPenn ignores all internal noun phrase structures, including the internal structures of multiple conjoined noun phrases. Hence, the bracketing is flat for the following phrase:

```
(NP the recent California earthquake and hurricane in the Carolinas)
```

Unfortunately, the parses which are produced by a parser system are more detailed than the treebank parses. The comparison of the parse to a skeleton representation renders a misleadingly low precision score. In Figure 2.4. the two spurious

```
Treebank:
[He [said [evidence [obtained [in [violation [of [the legal rights [of
citizens]]]]]]]]]

Parse:
[He [said [evidence [obtained [in [violation [of [[the [legal rights]]
[of citizens]]]]]]]]]

Spurious constituents in the parse:
  [the legal rights]
  [legal rights]
Recall  = 9/9 = 100.0%
Precision  = 9/11 = 81.81818181818183%
Crossing = 0
```

Figure 2.4: A sample analysis of the evaluation metrics

constituents [legal rights] and [the legal rights] are reasonable analyses, and a good

evaluation method should not penalize a parse with a finer granularity than the treebank.

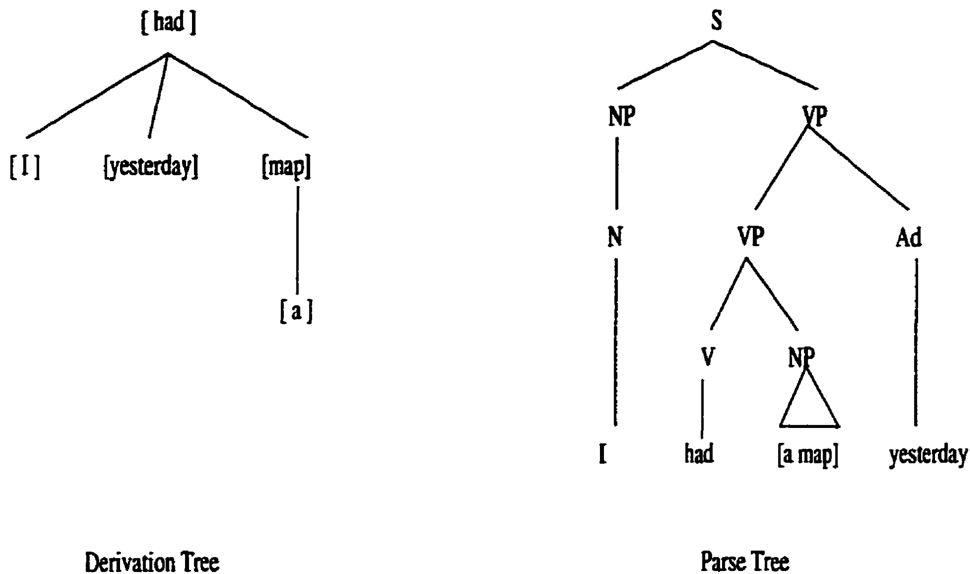


Figure 2.5: A sample derivation tree of XTAG and its phrase tree

Although most broad coverage parsers are constituency-based, a few broad-coverage parsers, such as XTAG [5], use derivation trees as the primary syntax structure, and the phrase structures are derived from derivation trees. A sample of a derivation tree and its phrase tree is shown in Figure 2.5 [5]. A derivation tree represents the derivation history of a parse and actually captures the dependency relations between words. Evaluating derivation trees would provide more direct measures of this type of parsers, than evaluating phrase trees.

Chapter 3

The Dependency Based Evaluation Method

With the emergence of broad-coverage parsers, the improvement of parser evaluation becomes an important task. According to the early studies of the parser evaluation [1, 12, 17, 2], an ideal evaluation criterion should fulfill the following requirements:

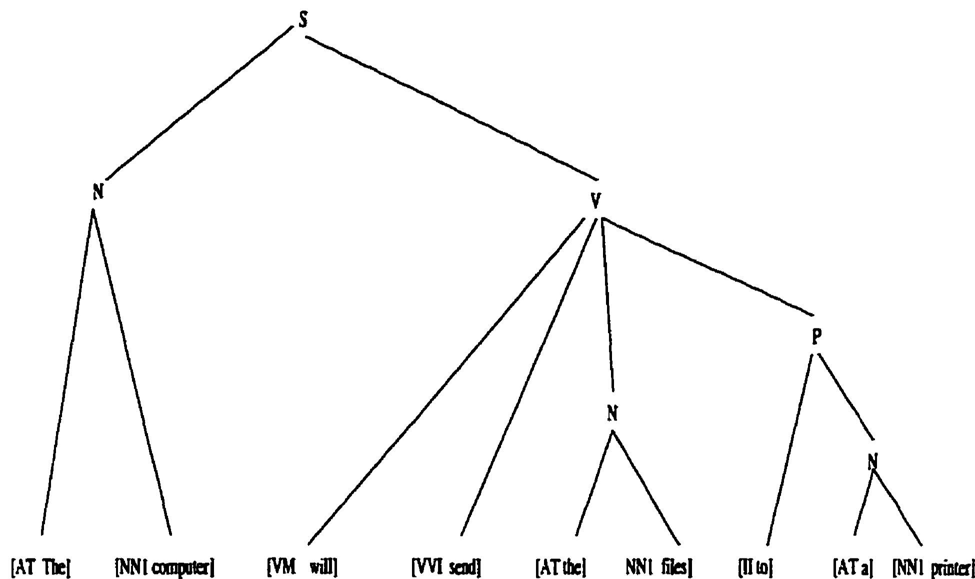
- An evaluation criterion must be based on the comparison with manually, or semi-automatically, created parses; the comparison must be conducted automatically, because of large volume of data.
- An evaluation method should not only tell us the degree of the performance of a parser, but also should focus on the errors, so that the evaluator can explain and remedy them.
- An ideal evaluation should be theory neutral; this means that an evaluation criterion should not have a prior bias towards any particular parser.

The development of the dependency-based evaluation method is largely motivated by these demands. This method uses dependency syntactic structures (also called dependency trees) as the formal syntactic representation in both treebanks and parser generated parses; therefore, a parser is scored by comparing dependency relations, while the constituency methods compare constituency boundaries.

3.1 Dependency Theory

This section details the dependency syntax, which the dependency-based evaluation method is based on. As is well known, there are two diametrically opposed methods in syntactic analysis: dependency syntax and constituency syntax. Constituency syntax, also known as "Phrase Structure Syntax", tends to insist on taxonomy, i.e., classification and distribution. Dependency syntax is based on relations between ultimate syntactic units, and therefore, it tends to be concerned with meaningful links, i.e. semantics [18]. Figure 3.1 shows the dependency syntactic structure and the constituency syntactic structure for the sentence: "The computer will send the files to a printer".

For a long time, constituency theories dominated English syntax theories and dependency theories received little attention from English linguists. However, the feasibility of dependency syntax in English has been recognized recently and the present researches in theoretical syntax have shown an increase of the interest in dependency syntax [18].



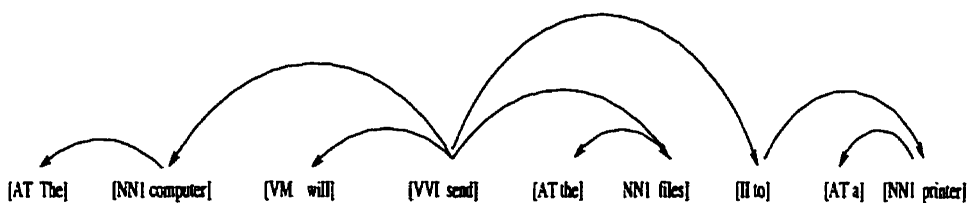


Figure 3.1: The dependency versus constituency

3.1.1 Dependency Syntax

In dependency theory, a word is both the smallest unit and the largest unit of the syntax. The only means to represent the syntactic structure of natural language sentences is binary directed syntactic relations between words. According to Melčuk [18], these relations should be:

- antisymmetric (or directed)

  if $X \rightarrow Y$, then $\neg(X \leftarrow Y)$. The direction of a relation enable us to distinguish a phrase pair like "minority student" versus "student minority"; we have a relation 'student → minority' for the first phrase, and 'minority → student' for the second.

- antireflexive

  $\neg(X \rightarrow X)$, since no word can be linearly affected under its own influence.

Another important feature of the relation is that it can be labeled according to its relation type, in order to distinguish one from another. In dependency theories, this kind of relations is called syntactic dependency relations. The terms, 'modifier' and 'head', are often used to refer to the two members of a dependency relation; 'modifier' depends on 'head', or conversely, 'head' governs 'modifier'. An important question here is how to identify one member of a relation as the head. The most general answer is that it is the head that provides the link between the modifier and the rest of the sentence, rather than vice versa [9]. Table 3.1 presents dependency analyses of some constructions.

The above observation leads to the following notion of syntactic structure: a set of words linked by syntactic relations. In addition, this syntactic structure must fulfill three conditions [9]:
For any well-formed natural sentence:

- One and only one word does not depend on another word. This word is the head of the sentence.

- All other words depend directly on some other words

modifier	head	relation type	example
complement	verb	complement	see(H) a man(M)
subject	verb	subject	I(M) see(H)
adjective	noun	adjunct	red(M) hat(H)
prep. obj.	preposition	complement	in(H) the park(M)
determiner	noun	specifier	the(M) park(H)

Table 3.1: Dependency relations of some constructions. M indicates modifier and H indicates head.

- No word depends directly on more than one other word
- Adjacency Principle: if word A directly depends on word B and word C is between A and B, then the head of C is A or B or some other word between A and B. The Adjacency Principle illustrates an important property in word-order.

In the mathematical sense, the syntactic structure of a natural sentence is a rooted tree in which nodes are the words of the sentence and arcs show the relationships among them. The root of the tree is the word which is the head of the sentence. Therefore, this syntactic structure is called a **Dependency Tree**.

3.1.2 Dependency versus Constituency

Dependency grammarians such as Hudson and Melčuk argued that dependencies are better suited for describing a syntactic structure rather than constituencies are. There are some advantages to use of syntactic dependencies [10, 18]:

- Words which need to be related directly to one another can be related in a dependency syntax. However, in a constituency syntax, phrase nodes usually intervene. For instance, in Figure 3.1, the relation between the verb 'send' and the preposition 'to', that the verb selects lexically, is a direct dependency. In constituency structure the verb 'send' is the sibling of phrase node 'P', and 'to' is the child of 'P'. The verb 'send' and the preposition 'to' have only an aunt-niece relation.

- Word-order rules can be formulated in a dependency more easily; especially in languages such as Japanese and Welsh, in which the head always follows or precedes its dependents.

- Syntactic dependency structures and semantic structures are very close. If word A depends on B in syntax, then A often semantically depends on B.

Although constituency is still the main syntax to be used in natural language processing, dependency syntax has become a serious alternative. Some recent developments in syntactic theory have shown an increase in the role of phrase head. One of the typical examples is the X-bar theory which is widely used in broad-coverage parsers. In the X-bar theory, every phrase is required to have a head which is used in almost the same way as in dependency theories. For example, in [$_{N}$ [$_{Det}$ a] [$_{Adj}$ cute] puppy [$_{P}$ with big ears]], the head is the word 'puppy'. The phrases or words [$_{Det}$ a], [$_{Adj}$ cute] and [$_{P}$ with big ears] are called modifiers of the head [$_{N}$ puppy]. The meaning of the phrase is largely determined by its head.

3.1.3 Syntax versus Semantics in Dependency Syntax

One merit of the dependency syntax is that its syntactic structure closely matches its semantic structure. In dependency relations, modifiers supply fillers for the **slots** in the semantic representations of heads [8]. Let us compare the semantic structure and dependency syntactic structure for the sentence: "John broke the window with a stone", which are shown in Figure 3.2. The semantic structure can be seen as the semantic representation of the verb 'break' which is a frame specifying a variety of slots. The modifiers 'John' and 'window' fill the agent and target slots, respectively. In addition, 'stone' and 'break' are linked (indirectly) by 'with'. Although the two structures are not completely identical, the syntactic dependency structure may be seen as at least very nearly in step with the semantic structure [9]. As can be seen in Figure 3.2, in the constituency syntactic structure, the verb 'break' is not directly related to the fillers of its semantic slots.

3.2 Dependency-based Evaluation

Unlike the constituency-based evaluation, the dependency-based evaluations adopts dependency trees to represent both treebank parses and parser generated parses. In the rest of this thesis, a treebank dependency tree will be called "a key" and a parser generated dependency tree will be referred to as "an answer". Evaluations are based on the comparison between the keys and their corresponding answers.

As mentioned in the last section, a dependency tree is a set of words which are linked by dependency relations. Thus, the evaluation score is computed in terms

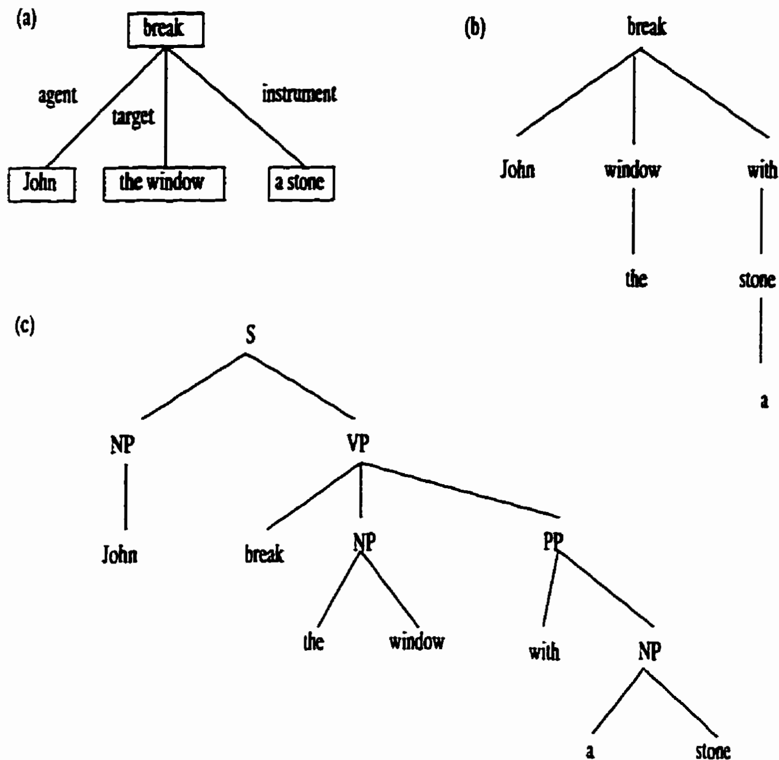


Figure 3.2: The semantic structure (a), dependency syntactic structure (b) and constituency syntactic structure (c) of the sentence: "John broke the window with a stone".

of the difference in the dependency relations between the key and answer on a word-by-word basis.

3.2.1 The Representation of Dependency Trees

To implement a dependency-based evaluation, a dependency tree should be represented by an appropriate data structure. The following hierarchy is used as the representation of a dependency tree:

- dep-tree: (word-slot, word-slot, ... , word-slot)

- word-slot: (word-token, wordsense)
- wordsense: (root, category, dependency-link)
- root: the root form of the word; e.g. the verb 'has' has the root form 'have'
- category: the part of speech tag of the word; i.e. N, V, and so on
- dependency-link: (head category position [relation type])
- dependency-link: `* or ?`
- position: `< or > or << or >> or ...`
- relation type: `specifier or complement or adjunct or subject or ...`

A dependency tree consists of a list of word-slots. Each word-slot specifies the dependency relation between one word and another word in the sentence, and has a word-token which is followed by a wordsense. The dependency link specifies the association between two wordsenses. The first and second elements in the dependency link are the word-token and the category of the head of the word.

The third element indicates the position of the head relative to this word. By definition, a position could be $\underbrace{<\cdots<}_{n}$ or $\underbrace{>\cdots>}_{n}$, where n can be any integer except 0. $\underbrace{>\cdots>}_{n}$ means the nth occurrence of the word "head" after the word, and $\underbrace{<\cdots<}_{n}$ means the nth occurrence of the word "head" before the word. A dependency link can also be `'*' or '?'`. `'*'` indicates that the word is the head of the sentence, and '?' indicates that the head of the word is unknown (or empty). The last element, "relation type", is optional.

The above representation is derived from the dependency tree representation in [12]. It uses a hierarchical structure instead of the flat structure in [12]. One reason for the change is that the hierarchical structure can be easily extended to accommodate the parse trees of ambiguous sentences, which will be discussed in Chapter 6.

For the sentence in Figure 3.1, Table 3.2 presents its sample tree which is used in the dependency-based evaluation.

```
(The      ('the'      AT    (computer NN1  <  spec)))
(computer ('computer' NN1   (send     VVI  <  subj)))
(will     ('will'     VM    (send     VVI  <  pred)))
(send     ('send'     VVI   (*)))
(the      ('the'      AT    (files    NNI  <  spec)))
(files    ('file'     NN1   (send     VVI  >  cmpl)))
(to       ('to'       P     (send     VVI  >  cmpl)))
(a        ('a'        AT    (printer  NN1  <  spec)))
(printer  ('printer'  NNI   (to       P    >  cmpl)))
```

Table 3.2: The dependency structure of the sentence: "The computer will send the files to a printer"

3.2.2 Evaluation Metrics

Once the key and answer are both represented as the dependency structures, the evaluation can be conducted by comparing the word-slots in the answer to the corresponding word-slots in the key, one-by-one.

In the dependency-based evaluation, the primary evaluation measure for system performance is the error rate and the secondary metrics are recall and precision. The computation of these measures depends on the types of word-slots in answers. The word-slots in an answer can be classified as follows:

- correct: the dependency link of the word-slot is equal to the dependency link of the corresponding word-slot in key
- incorrect: the dependency link of the word-slot does not match the dependency link of the corresponding word-slot in key
- missing: the dependency link of the word-slot is empty and it is filled in key
- spurious: the dependency link of the word-slot is filled and it is empty in key
- noncommittal: the word-slots both in the answer and in the key have empty dependency link

There are two modes to determine if two dependency links are equal: the general mode and the exact match mode. In the general mode, two dependency links are equal when they have the same word-token and position value (in the other words, they point to the same head). However, for two equal dependency links in the exact match mode, their relation types should match, as well as their word-tokens and positions.

The error rate

Since a dependency tree can be considered as a sequence of discrete elements, Lin has proposed the use of the Hamming Distance to describe how close the answer and the key are to each other [12]. The following is the definition of Hamming

Distance:
For any two corresponding word-slots in the answer and the key, the Hamming distance between the word-slots is the minimal number of steps of operations needed to make one slot equivalent to another one.

Three operations are defined for computing the Hamming Distance:

- addition: add any dependency link to a word-slot
- deletion: delete any dependency link from a word-slot
- substitution: replace a dependency link in a word-slot with another one

The value of the Hamming Distance between two slots can be 1 or 0. In addition, for a specific slot in an answer, an error count is defined as the Hamming Distance between the slot and the corresponding word-slot in the key. Any missing, spurious, and incorrect word-slots will be counted as 1, and any correct and noncommittal word-slots will be counted as 0. The error count of an answer is the sum of the Hamming Distance between each word-slot in the answer and its corresponding slot in the key.

Thus the error rate is calculated as follows:

$$error\ rate = \frac{the\ error\ count\ of\ answers}{the\ total\ number\ of\ words\ in\ answers}$$

Recall and precision

Besides the Hamming Distance-based measure used in [12], recall and precision are also added to the metrics. The concept of the recall and precision metrics

was adapted from the DARPA parser evaluation method. The use of recall and precision has two advantages:

- They measure two different aspects of a performance: completeness (recall) and accuracy (precision).
- They present a positive view of system performance, which encourages parser researchers to submit their systems for evaluation.

The formulas to compute recall and precision are as follows:

$$precision = \frac{the\ total\ number\ of\ correct\ slots}{the\ total\ number\ of\ correct,\ spurious\ and\ incorrect\ slots}$$

$$recall = \frac{the\ total\ number\ of\ correct\ slots}{the\ total\ number\ of\ correct,\ missing\ and\ incorrect\ slots}$$

To compare the answer and its key in Figure 3.3, the scores of the answer are:

```
error rate = 5/9 = 0.56
recall = 4/9 = 0.44
precision = 4/6 = 0.67
```

One controversial point in the error rate metric is how to handle incorrect errors. Error rate treats an incorrect error the same as the spurious and missing ones. It is argued that recall and precision view "incorrect" as a blend of "missing" and "spurious" [3]; a parser did not simply produce the wrong dependency link, but also produced a spurious link on the one hand and a missing link on the other hand; therefore, the error rate metrics should view the system in the same way as recall and precision did. To be consistent with recall and precision, an alternative is that only two operations are used to compute the Hamming Distance: addition and deletion. Thus, the error count for an incorrect slot is 2, since one substitution must be replaced by one deletion and one addition.

```
key:
   (The       ('the'       AT      (computer  NN1  <   spec)))
   (computer  ('computer'  NN1     (send      VVI  <   subj)))
   (will      ('will'      VM      (send      VVI  <   pred)))
   (send      ('send'      VVI     (*)))
   (the       ('the'       AT      (files     NNI  <   spec)))
   (files     ('file'      NN1     (send      VVI  >   cmpl)))
   (to        ('to'        P       (send      VVI  >   cmpl)))
   (a         ('a'         AT      (printer   NN1  <   spec)))
   (printer   ('printer'   NNI     (to        P    >   cmpl)))

answer:
   (The       ('the'       AT      (computer  NN1  <   spec)))
   (computer  ('computer'  NN1     (will      VM   <   subj)))   incorrect
   (will      ('will'      VM      (*)))                         incorrect
   (send      ('send'              (?)))                         missing
   (the       ('the'       AT      (files     NNI  <   spec)))
   (files     ('file'      NN1     (?)))                         missing
   (to        ('to'        P       (?)))                         missing
   (a         ('a'         AT      (printer   NN1  <   spec)))
   (printer   ('printer'   NNI     (to        P    >   cmpl)))
```

Figure 3.3: The answer and key for the sentence: "The computer will send the files to a printer".

3.2.3 Selective Evaluation

Evaluation users may be interested in some of the specific syntactic structures. The dependency-based evaluation method can selectively evaluate the performance of parsers in regard to those structures. The idea of selective evaluation is to compute metrics in terms of conditions. Only the word-slots which match the condition are compared.

A condition can be a logical expression. Besides 'and', 'or' and 'not', an expression can have other operators such as 'type'. A word-slot is tested to be true by 'type X', if its dependency link has the relation type X. For example, if a word-slot matches '`(or (type complement) (type adjunct))`', its dependency link is of the type 'complement' or 'adjunct'. The evaluation algorithm is given as follows:

```
evaluate(answer, key, condition)
{
    for each word in the sentence{
       Let K be its word-slot in key and A be its word-slot
        in answer
        if K or A satisfies the condition{
             if A is correct slot
                 #_of_correct = #_of_correct + 1
              if A is incorrect slot
                 #_of_incorrect = #_of_incorrect + 1
              if A is missing slot
                  #_of_missing = #_of_missing + 1
              if A is spurious slot
                 #_of_spurious = #_of_spurious + 1
        }
     }
     error_count = #_of_incorrect + #_of_missing + #_of_spurious
```

```
    precision = #_of_correct/(#_of_correct + #_of_spurious + #_of_incorrect)
    recall = #_of_correct /(#_of_correct + #_of_missing
               + #_of_incorrect)
}
```

Therefore, if we want know how a parse handles the subject relation, we can assign '(type subj)' to the logical expression of *evaluate.*

3.2.4 Transforming Constituency Trees into Dependency Trees

Since most broad-coverage parsers use the constituency syntax, it is necessary to find a way to transform constituency trees into dependency trees, in order to apply this method to a constituency-based parser.

The dependency tree and the constituency tree can be transformed into each other by a mechanical procedure, which involves some reorganization [8]. Converting a dependency tree to a constituency tree is straightforward. For each word in a dependency tree, a phrase consists of the word plus all the words that modify it. However, an additional mechanism is required to represent this information to identify the head of a phrase. The rest of the section will explain the conversion from constituency trees into dependency trees.

Tree Head Table

The Tree Head Table is proposed by Magerman [17] to assign lexical heads to the constituents for a P-CFG parses. A Tree Head Table for a specific grammar is a set of deterministic rules which have the following syntax:

`(parent-label direction a-list-of-category-tags-and-labels)`

where a **`parent label`** is the label of the constituent whose lexical head is being assigned; **`direction`** indicates whether or not the children of this constituent are processed left-to-right or right-to-left; the remainder of each rule consists of an ordered list of part-of-speech tags and constituent labels, which might occur as the children of the constituent. The priorities of tags or labels in the list decrease from left to right. In addition, any tag or label appearing in the list has a higher priority value than those which are not in the list. The lexical head of a constituent is identified as the lexical head of the child whose label (or tag, if the child is not a constituent) has the highest priority. In the case of two children having the same priority level, if the **`parent label`** is marked as left-to-right, then the leftmost one is selected; otherwise, the rightmost one is selected.

```
rule 1 : S right-to-left S V Ti Tn Tg N J Fa REX22 ...
rule 2 : N right-to-left N NNJ  NNU .., NN1  ...
rule 3 : V left-to-right V ... VVI ..... VM ...
```

Figure 3.4: A subset of Magerman's Head Table.

The example in Figure 3.5 illustrates this process. Based on a subset of the rules in Magerman's Tree Head Table in Figure 3.4, the assignment algorithm works bottom-up. For [$_N$ The computer], the noun 'computer' has a higher priority than 'the' since NN1 is in the right hand list of rule2, but AT is not. For the same reason, [$_N$ the files] has the lexical head 'file', and [$_N$ a printer] has the lexical head 'printer'. Now, consider [$_V$ will send the files to a printer]. In rule 3, VVI appears on the left side of VM; therefore 'send' is selected as the lexical head. Finally, since

the priority of N is lower than that of P in rule 1, 'send' becomes the head of the sentence.

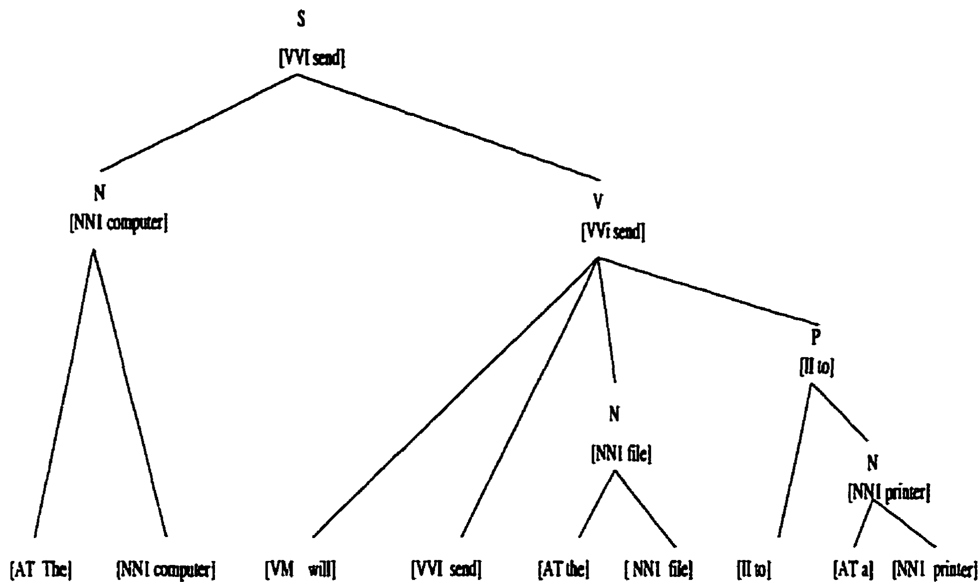


Figure 3.5: An example for the Head Assignment algorithm

After the assigning of a lexical head to each constituent in a parse tree, dependency relations can be derived as follows: the lexical head of a constituent governs the lexical head of its child constituent or its child word, if the child is not a constituent.

Relation Table

To add the type of dependency relations during derivation, a Relation Table is used. Figure 3.6 shows an example of a relation table.

Like the Tree Head Table, a Relation Table includes a set of rules. The left hand

side of rules is the tag of the lexical head of a constituent, and the right hand side is a set of triples in which the first element is the label of the constituent, the second is the tag of the modifier of the lexical head, and the last one is the type of relationship that could be assigned to the modifier-head pair.

```
NN1 ---- (N AT spec)
V   ---- (S NN1 subj)(VP NN1 cmpl)(VP P cmpl)
II  ---- (P N cmpl)
```

Figure 3.6: A example of a Relation Table

The first rule in Figure 3.6 means that if the tag of a lexical head is NN1, the constituent to which the lexical head is assigned is N, and a modifier of the head is AT, then the type of dependency relation between the lexical head and its modifier is "spec". Based on the above relationship table and the constituent tree in Figure 3.4, the dependency relations can been presented in Table 3.3.

Conversion from PRINCIPAR Parse Trees to Dependency Trees

Most of modern linguistic formalisms, such as the Government-Binding Theory, have the notion of the head of a constituent. For parses that are created with such theories, no extra effort is needed to identify the head of constituents.

For example, in the parse trees produced by PRINCIPAR [9], a constituent is represented in the XP structure which is given in Figure 3.7. X is a variable and could be N, I, P, etc. In an XP, the X represent the head of a constituent. The

MODIFIER	HEAD	RELATION TYPE
The	computer	spec
computer	send	subj
will	send	pred
the	files	spec
files	send	cmpl
to	send	adjn
a	printer	spec
printer	to	cmpl

Table 3.3: The dependency relations of the sentence: "The computer will send the files to a printer".

complements, adjuncts and specifiers are the modifiers of the head.

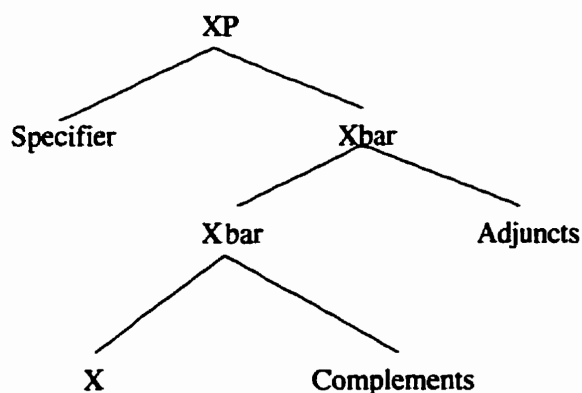
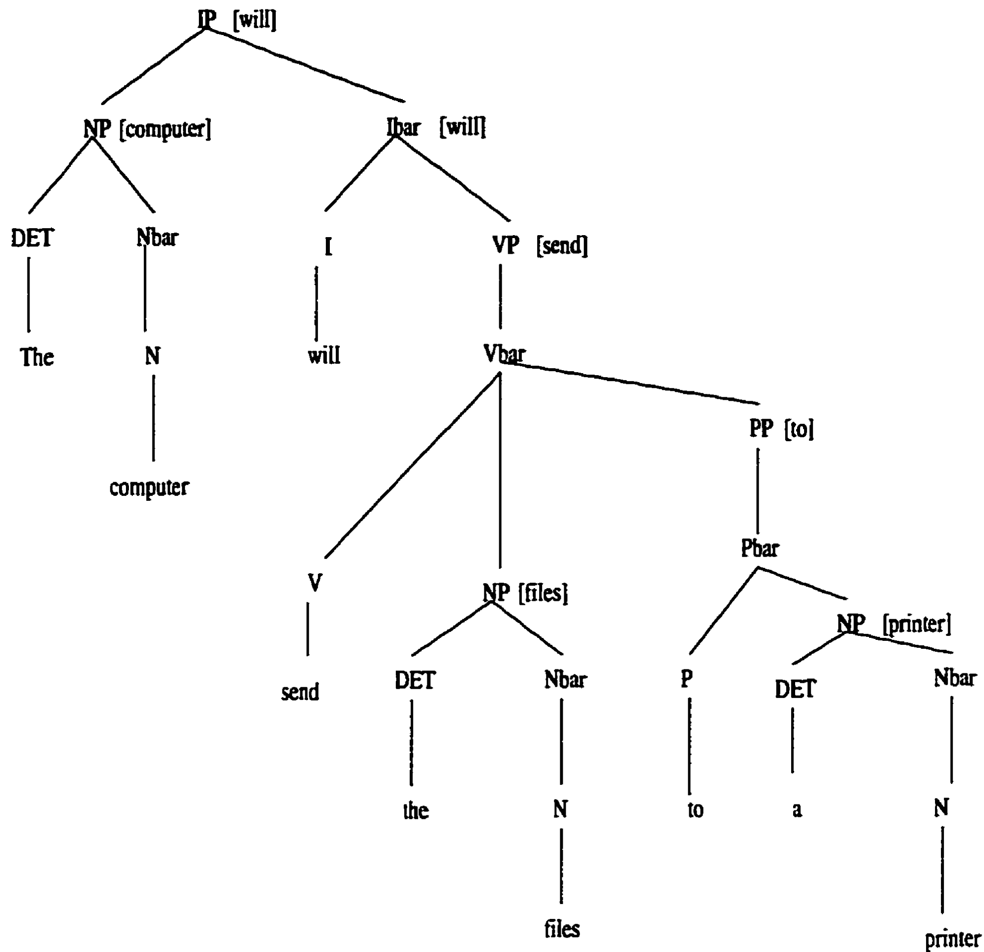


Figure 3.7: The structure of XP

For the same sentence: "The computer will send the files to a printer", Figure 3.8 shows its parse tree and the derived dependency tree.



MODIFIER	HEAD	RELATION TYPE
The	computer	spec
computer	will	subj
send	will	pred
the	files	spec
files	send	cmpl
to	send	adjn
a	printer	spec
printer	to	cmpl

Figure 3.8: The X-bar tree and derived dependency relations of the sentence "The computer will send the file to a printer".

3.2.5 Advantages of the Dependency-Based Evaluation

Some advantages of dependency-based parser evaluation are as follows:

1. According to [12], the metrics are intuitively meaningful, since the semantic dependency is embedded in the syntactic dependency. Therefore, the metrics are more relevant to how useful a parse is than the metrics in the DARPA evaluation.

2. In the DARPA evaluation method, a system may be ranked low precision, if it has a finer granularity than the treebank. However, the dependency-based evaluation is less sensitive to the granularity of a parse tree. For the example of Figure 2.4, two analyses [the legal rights] and [the [legal rights]] result in the same dependency structure (see Figure 3.9). Thus such differences are ignored by the dependency-based evaluation.

Figure 3.9: The dependency structures of [the legal rights] and [the [legal rights]]

3. Error rate itself is an error based metric, and it helps parser researchers focus on errors which a parser has made. Furthermore, a selective evaluation of a particular type of phenomena is easy to apply to a dependency tree, and it will provide much useful information to analyze the performance of a parser.

4. In the XTAG parser, which is described in the last chapter, a derivation tree is equivalent to a dependency tree, except that it is unlabeled. This method provides a direct measure to evaluate the derivation structure, instead of its secondary

phrase structure.

Chapter 4

A Modification Tool for Dependency Trees

The function of the modification tool is to manipulate dependency trees by principles which are defined by evaluators. This chapter explores why dependency trees in parser evaluation need to be modified, and explains how the modification tool works.

4.1 The Objective

One of the biggest problems encountered by parser evaluators is the difficulty of defining standard parse trees for measuring the outputs of the different parsers. In Table 3.3 and Figure 3.8 for the sentence: "The computer will send the files to a printer", two dependency trees treat the dependency relation between the auxiliary 'will' and main verb 'send' differently. In Magerman's Tree Head Table, [$_{VVI}$ send] is considered to have a higher priority than [$_{VM}$ will]; therefore, 'will' is the modifier

of 'send'. But the PRINCIPAR selects 'will' as the lexical head and 'send' depends on 'will'. Both analyses are valid within their own theories.

English grammars appear to disagree strongly with each other as to the elements of even the simplest sentences. For example, while it is generally accepted that the main verb of a sentence governs its subject, some grammarians also argue that the verb may depend on its subject, since the subject controls the form of the verb [18]. Other differences include the treatment of the conjunction, 'not', pre-infinitival 'to', and so on.

An ideal evaluation method should allow discrepancies among grammatical theories, and only measure parsers according to each theory, not prefer one and discriminate against others.

In the DARPA evaluation method, the elements involving some controversial structures are erased. However, this affects the accuracy of evaluation, and not all such phenomena can be eliminated. In contrast, the dependency-based evaluation provides a modification tool which is able to transform one dependency structure into another, by following pre-defined rules. Before the evaluation of dependency trees, each dependency tree is revised so as to remove the allowable differences among the parses.

4.2 Rule

The modification tool provides rules to transform dependency trees. The concept of the rules is derived from the modifying operations in [12]. While the modifying operation is used to modify a dependency link, the rule is extended to modify a multi-layer sub-dependency tree.

Each rule constitutes a "pattern" part and an "action-list" part. The syntax of rule is:

```
(
  (IF pattern)
  (THEN action-list)
)
```

The function of a rule is to search a dependency tree to find any subtree to match the pattern and modify it, by the actions in the action-list.

4.2.1 Pattern

The pattern is implemented as a tree. The node of a pattern tree contains a logic expression which is called a node-expression. The node-expression is used to evaluate the word-slots in a dependency tree. The arc of a pattern tree is labeled by another logic expression which is called link-expression. The function of the link-expression is to evaluate the dependency links in a dependency tree.

Besides the relational operators 'and', 'or' and 'not', the expressions can have four other operators 'cat' , 'string', 'type', and 'pos'. The first two operators are used

in a node-expression and, 'type' and 'pos' are for link-expressions. These operators are defined as follows:
if X is a regular expression, then

- (cat X) is true if the evaluated word-slot has a category to match X
- (string X) is true if the word-slot has a root form to match X
- (type X) is true if the dependency link has a relation type to match X
- (pos Y) : Y can be 'pre' or 'post'
  (pos pre) is true if the head is before the modifier in the dependency link
  (pos post) is true if the head is after the modifier in the dependency link

A subtree matches a pattern, if and only if:

- the subtree has the same configuration as the pattern tree
- each word-slot W in the subtree is evaluated to be true by the node-expression of the corresponding node N in the pattern. If W is not the root of the pattern, then the dependency link in W is evaluated to be true by the corresponding link-expression in the pattern.

In the modification tool, a pattern can be easily represented by a LISP-like list, such as:

```
(root
    (first_child
        (first_grandchild second_grandchild...)
     second_child
      ...
    )
 )
```

where each node in the list contains its node-expression, as well as the link-expression of the arc between the node and the parent of the node. The following is the grammar of pattern:

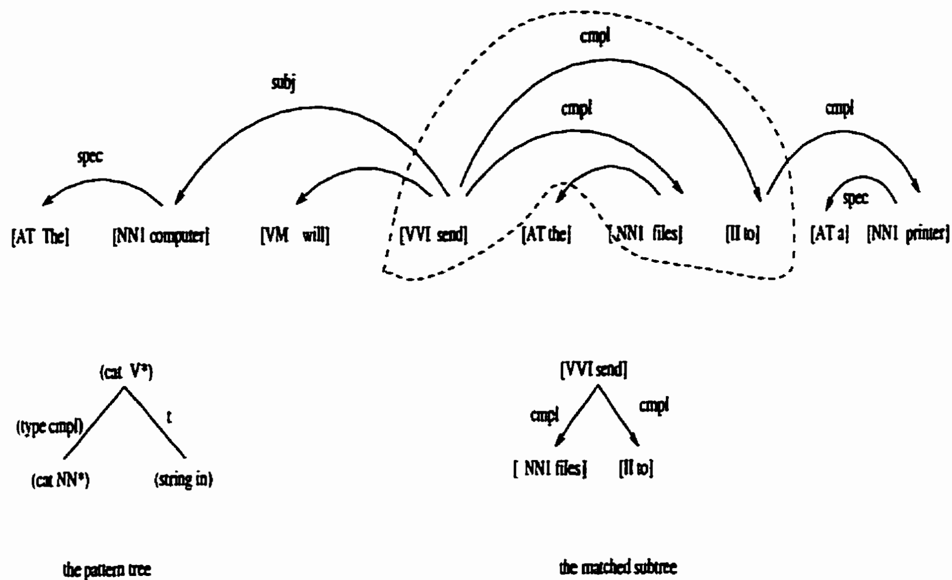
```
Terminal-symbols={NODE-EXP, LINK-EXP, regular-expression, and, or,
    not, cat, string, type, pre, post, t, ), (}
Nonterminal-symbols={Pattern, Root, Forest, Subtree, Node, Node-expression,
    Link-expression}

Pattern -> (Root (Forest))
Forest -> Subtree Forest | Subtree
Subtree -> Node (Forest) | Node
Root -> (NODE-EXP Node-expression)
Node -> ((NODE-EXP Node-expression)(LINK-EXP Link-expression))
Node-expression -> (not (Node-expression))
                  |(and (Node-expression)(Node-expression))
                  |(or (Node-expression)(Node-expression))
                  |(cat regular-expression)
                  |(string regular-expression)
                  | t
Link-expression -> (not (Link-expression))
                  |(and (Link-expression)(Link-expression))
                  |(or (Link-expression)(Link-expression))
                  |(type regular-expression)
                  |(pos pre) | (pos post)
```

Figure 4.1 shows a sample of a pattern and its match.

4.2.2 Action

An action is a modification to dependency relations such as 'isolate', 'delete', 'convert', and 'transfer'. The arguments for an action specify the relative position of




```
pattern: ((NODE-EXP (cat V*))
              (((NODE-EXP (cat NN*))(LINK-EXP (type cmpl)))
               ((NODE-EXP (string in)) (LINK-EXP t))
              )
          )
```

Figure 4.1: A pattern and its match

nodes in a subtree. Its syntax is a sequence of numbers separated by a dot. For example, 0 represents the root of a subtree; 0.2 represents the second child of the root and so on. The following actions have been defined:
(delete X Y): removes the dependency link between X and Y, where X is the head of Y

(singleTransfer X Y Z): when Z is the modifier of Y and Y is the modifier of X, the action transfer Z to X. Thus Z becomes the modifier of X. The action includes the following steps:

- remove the dependency link between Y and Z
- add the dependency link between X and Z

(transfer X Y): is like singleTransfer, but it transfers all modifiers of Y to X
(invert X Y): when a dependency relation (X, Y) is inverted, the modifier Y becomes the head of the head X. In meanwhile, the head of X becomes the head of Y

- reverse the dependency link between X and Y, where X is the head and Z is the modifier
- remove the dependency link between the head of X and X
- if X is not the head of the sentence, add the dependency link between the head of X and Y
- if X is the head of the sentence, Y becomes the head of the sentence after the direction is reversed.

(isolate X): The action removes a node which is not the head of sentence from dependency tree. The following steps take place while executing "isolate":

- (transfer the head of X X)
- (delete the head of X X)

The remainder of this section uses a few examples to illustrate how the modification rule works. It is mentioned earlier that there may be two different analyses for the dependency relation between the auxiliary and the main verb. Those different analyses can be transformed into each other by using the rules specified in Figure 4.2.

Let us consider the coordinate structure which is another controversial structure in dependency theories. Figure 4.3 illustrates two possible dependency trees of the sentence: "He stood up and gave me the letter". We are able to transform one dependency tree to another, as shown in Figure 4.3.

Figure 4.4 shows another example. In the adverb clause 'If you invite me', some grammars treat the verb 'invite' as the head of the clause; however, other grammars may consider that 'if' should be the head of 'invite'. The rules in Figure 4.4 can eliminate the discrepancy among those grammars.

The action "isolate" provides a similar function to the erasure in the DARPA evaluation method. For example, in Figure 4.5 , pre-infinival 'to' can be either the modifier of 'want', or the modifier of 'do'. Using the rule:

```
rule A
(
  (IF ((NODE-EXP t)
        (((NODE-EXP (cat TO))(LINK-EXP t))
```

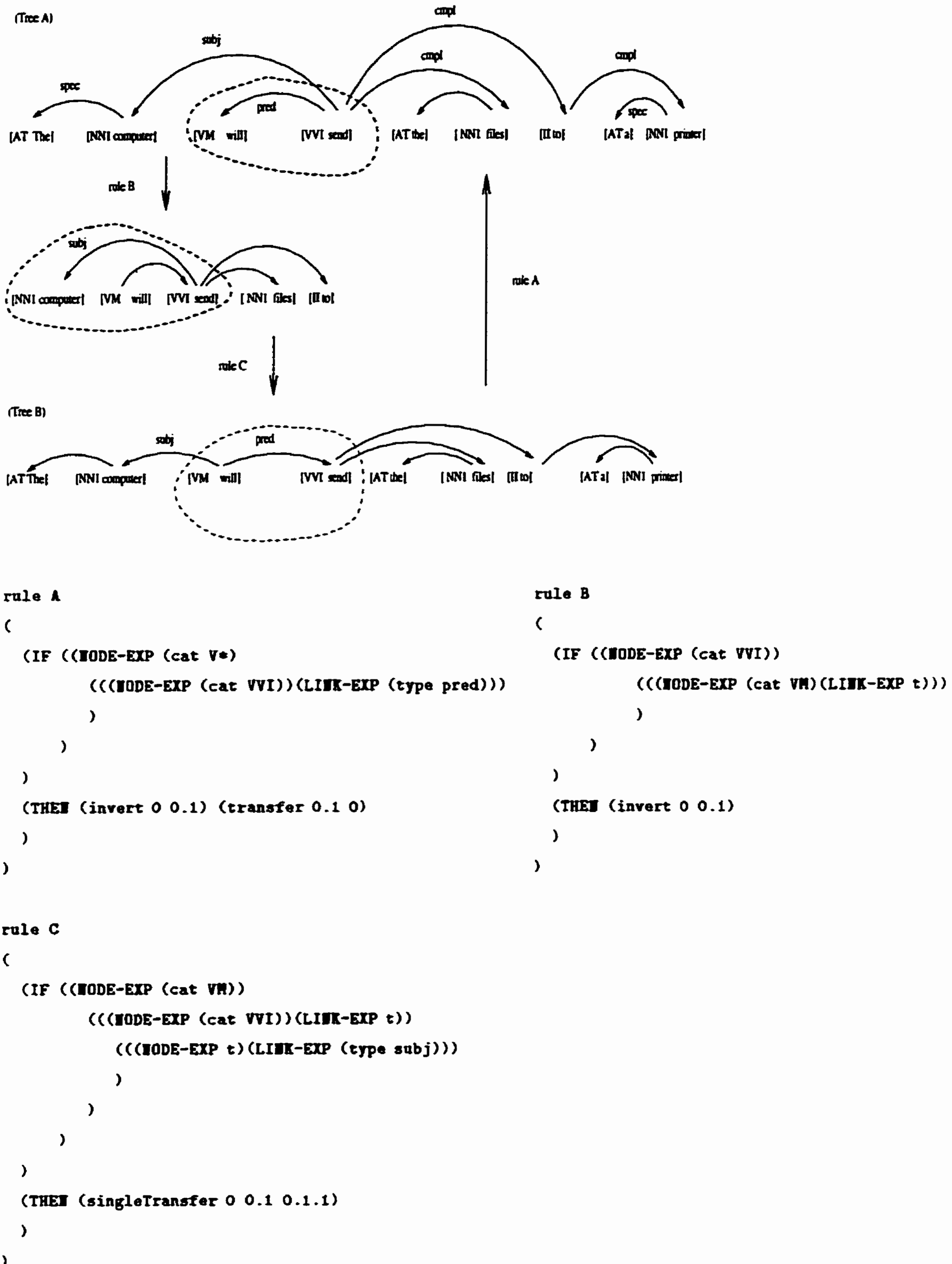


Figure 4.2: Using rules to transform (a) to (b) and vice verse. The structures circled by dotted line are the matches of the patterns of the corresponding rules.

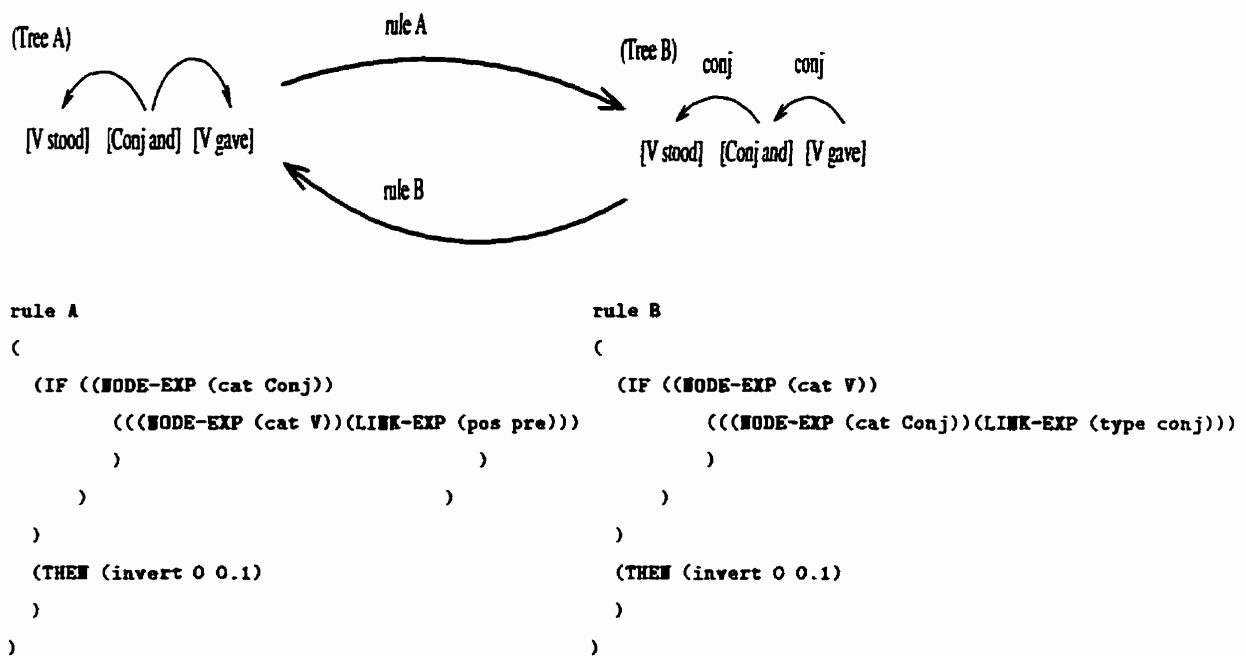


Figure 4.3: Transform different analyses of coordinate structures. Tree A and Tree B are two ellided dependency tree for the sentence: "He stood up and gave me letter".

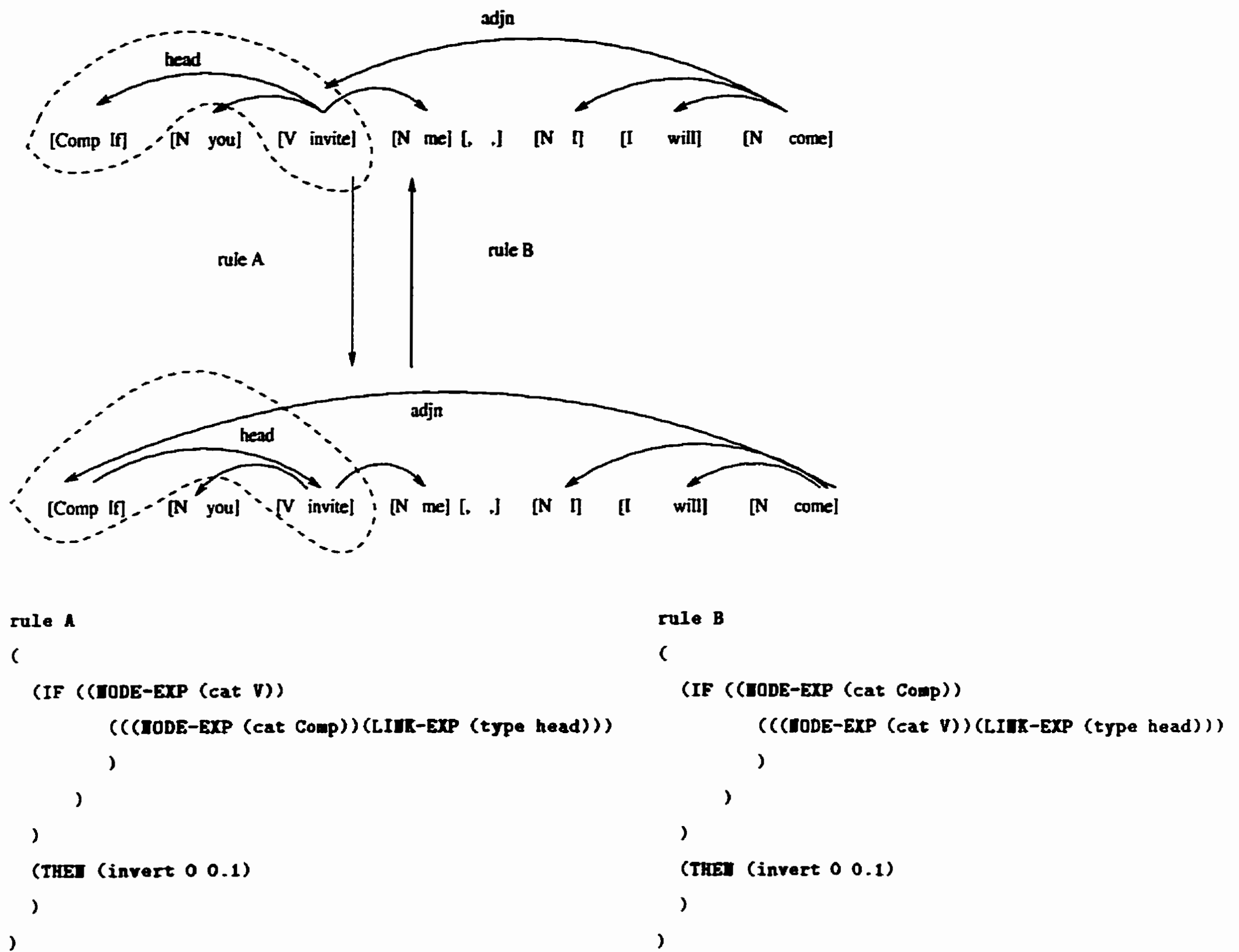


Figure 4.4: One example of the dependency tree modification

```
        )
      )
  )
  (THEN (isolate 0.1)
  )
)
```

the two structures are transformed to an identical form,

A set of rules, called a modification module, can be defined for a particular grammar. For an input parse, each rule is fired in a sequence. Since a rule may produce new subtrees which match the pattern of the rule or other rules, the process of modifying is repeated until each rule in the module can not find a matched subtree.

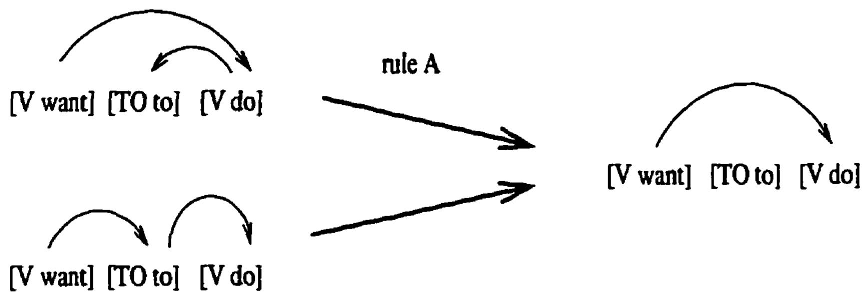


Figure 4.5: Normalization of pre-infinival 'to'

The modification tool can also be applied to other natural language processing systems. If the systems require a conversion from relating dependency syntactic structures to semantic structures, it is possible to make use of the rules to formulate the general conversion. For example, in Figure 3.2, the filler of the 'instrument' slot of the verb is not related to the verb by a dependency link. The following rule removes the preposition 'with', so that a direct relation can be built between the verb and its filler:

```
(
  (IF ((NODE-EXP t)
         (((NODE-EXP (cat P))(LINK-EXP (type adjn)))
         )
      )
  )
  (THEN  (isolate 0.1)
  )
)
```

Chapter 5

Implementation and Experimental Results

The dependency base evaluation system is implemented by object-oriented methodologies. As is well known, object-oriented methodology focuses initially on the data that a system manipulates to do its job. In the evaluation system, such data are dependency trees, word slots, dependency links and so on.

Two basic concepts in object-oriented methodologies are object and class. An object is a concrete entity that has attributes and behaviors. A class represents an abstraction of objects that share a common structure and common behaviors. In an Object-Oriented programming language, a class is a data structure whose physical format is hidden behind a type definition. It embodies a set of formal properties (or attributes) and is manipulated by a set of methods (or operations).

One of the features of object-oriented method is abstraction [19]. In contrast with

procedural abstraction, the abstraction on which Object-Oriented technology is based is data abstraction. This is the key to the method's success in ensuring extendibility and reusability.

5.1 Class DependencyTree and EvaluationMetrics

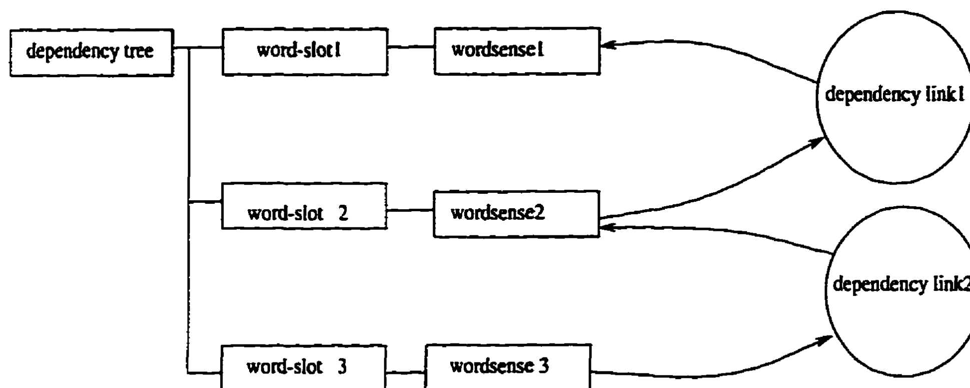


Figure 5.1: The architecture of the dependency tree

There are four main abstractions in the structure of dependency trees: dependency tree, word-slot, wordsense, and dependency link (see Figure 5.1); therefore there are four classes called DependencyTree, WordSlot, WordSense, and DependencyLink. The simplified class templates are shown as follows:

```
class DependencyTree
attributes:
      wordlist : a list of WordSlot objects

class WordSlot
attributes:
      wordtoken : a string which contains the word-token of the word
      wordsense : a pointer to a WordSense object
```

```
class WordSense
attributes:
        root : a string which contains the root of the word
        category : a string which contains the category of the word
        dependencylink : a pointer to the DependencyLink object of the word

class DependencyLink
attributes:
        relationtype : a string which contains the type of the dependency link
        head : a pointer to a WordSense object which is the head of dependency
                link
```

The object model of these classes, which uses Rumbaugh OMT notation [7], is shown in Figure 5.2. WordSlot is a part of DependencyTree and WordSense is a part of WordSlot. DependencyLink is an association between two WordSense objects.

The evaluation metric concept is implemented by the class EvaluationMetrics. Its operation *evaluate* compares two objects of DependencyTree and yields evaluation results.

5.2 Tree Modifying Algorithm

It was mentioned in the previous chapter that the dependency tree modification is based on rules which search all matches in dependency trees in terms of rule patterns and modify those matches by actions. This section presents a Tree Modifying Algorithm. We make use of two following definitions in the algorithm. Let X be a node in a rule pattern:

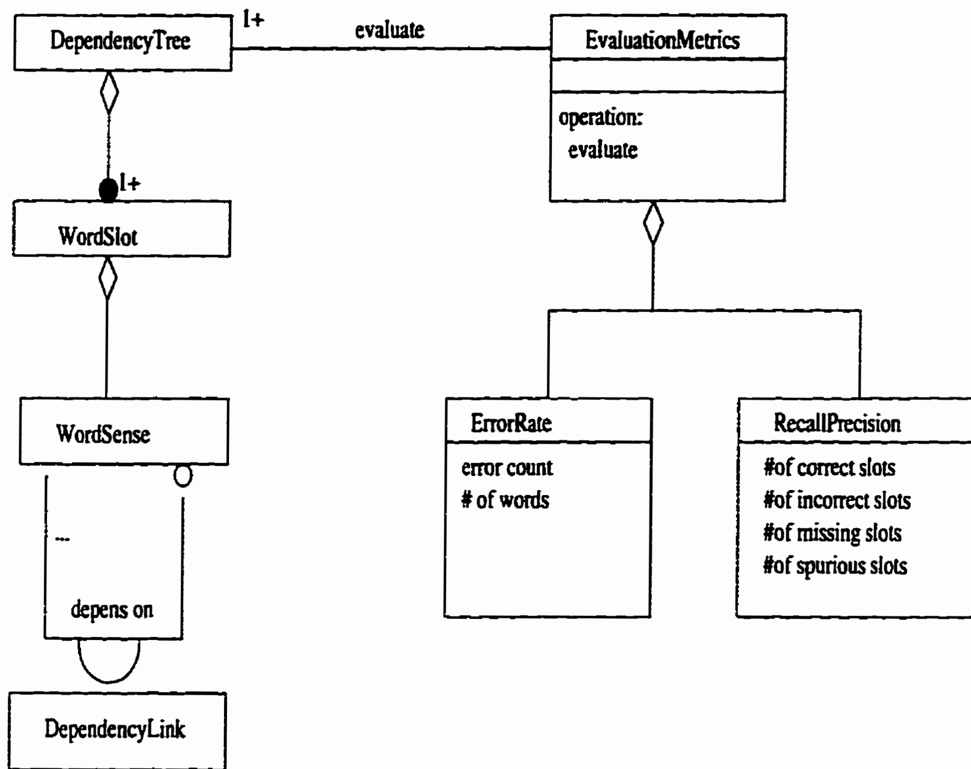


Figure 5.2: The Object Model which uses Rumbaugh OMT notation (see Appendix B)

- **linkMatch**: the linkMatch of X is a node N (or called word-slot) in a dependency tree which matches the node-expression of X, the dependency link of N matches the link-expression of the branch between X and the parent of X, and the head of N matches the node-expression of the parent of X. Note that the root of a pattern does not have linkMatch.

- **subtreeMatch**: the subtreeMatch of X is a subtree L of a parse tree which meets the following conditions:

  - if X is a leaf node, its linkMatch is its subtreeMatch.
  - each child of X has one subtreeMatch which is a child of the root of L.
  - if X is not the root of the pattern tree, the root of L is a linkMatch of X

  In other words, the subtree L matches the subpatterntree which includes X and all pattern tree nodes under X. In addition, the root node of L should be the linkMatch of X when X is not the root of pattern tree.

Let us use the dependency tree and the rule pattern in Figure 5.3 as an example. The linkMatches and subtreeMatches of the pattern nodes are given as following, trees are represented in LISP-like lists.

```
P4's linkMatch    : { D8, D2 }
     subtreeMatch : { (D8), (D2) }
P3's linkMatch    : { D5 }
     subtreeMatch : { (D5 (D8)) }
P2's linkMatch    : { D2, D4, D8 }
     subtreeMatch : { (D2), (D4), (D8) }
P1's subtreeMatch : { (D1 (D2 D5 (D8))), (D1 (D4 D5 (D8)))}
```

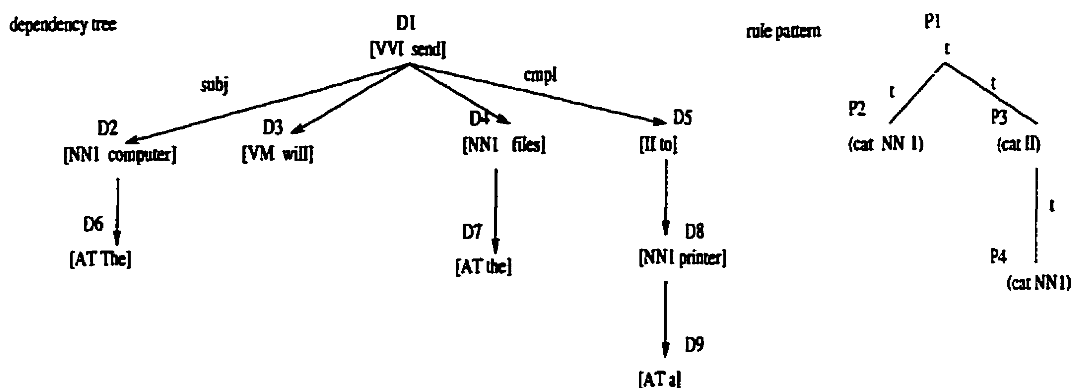


Figure 5.3: The dependency tree of the sentence: "The computer will send the files to a printer" and a rule pattern. D1, D2, ... are used to identify the nodes of trees.

To simplify the algorithm, we also make the following assumption: the rule writers have to ensure that if the node in a dependency tree is the linkMatch of X, the node can not be the linkMatch of the X's sibling nodes.

The main idea of the algorithm is that each pattern node makes use of two local memories to store the information about all its linkMatches and subtreeMatches for an input dependency tree. The two memories are called *LinkMatchList* and *SubtreeMatchList*, respectively. This information can be looked up to avoid computing these matches more than once.

5.2.1 Build initial matches

For a dependency tree, a rule generates the initial *LinkMatchList* and *SubtreeMatchList* of its pattern nodes from bottom to up. After a node finds its all linkMatches, the subtreeMatches of the node (except leaf nodes) can be computed in terms of the node's linkMatches and its children's subtreeMatches as indicated in the following

algorithm.

Algorithm 1(node X computes its subtreeMatches)

compute_subtreeMatch

n is # of X's children;

S_i is *SubtreeMatchList* of the ith child of X's; such as $\{t_{i1}, t_{i2}, ..., t_{ij_i}\}$,

where each element is the subtreeMatch of the child;

for each t in S_1

if X is not the root or *parent_t* is a X's linkMatch

$t_1 = t$

$T = \{t_1\}$

$k = 2$

recursive_search(k, T)

recursive_search(k, T)

if there is no t_k in S_k which is $t_1's$ brother

clear T and return stop_search

for each t_k in S_k which is $t_1's$ brother

add t_k to T

if $k = n$

subtreeMatch=make_tree(T)

save *subtreeMatch* into *X's SubtreeMatchList*

remove t_k from T and return

if recursive_search($k + 1, T$)=stop_search

return stop_search

remove t_k from T

return

make_tree(T)

concatenate each t in T to build a larger subtree

5.2.2 Apply actions

Each action of a rule can be decomposed into a sequence of removing or adding a dependency link. When each dependency link is added to or removed from the dependency tree, a rule updates the local memories of its pattern nodes at the same time, as shown in the following algorithm.

Algorithm 2 (add a dependency link)
postiterate each pattern node except the root
  $n = modifier_of_dependencylink$
  if n is a linkMatch
    save n to $LinkMatchList$
    $list_of_subtrees$ = compute_subtreeMatch(n)
    add $list_of_subtrees$ to $SubtreeMatchList$
    if $list_of_subtrees$ is not empty
      send $list_of_subtrees$ to its parent

Algorithm 3 (parent node receive $list_of_subtrees$)
  $list_of_subtrees'$=compute_subtreeMatch($list_of_subtrees$)
  add $list_of_subtrees'$ to $SubtreeMatchList$
  if $list_of_subtrees'$ is not empty and node is not the root
    $list_of_subtrees=list_of_subtrees'$
    send $list_of_subtrees$ to $parent_node$

In the algorithms above, the functionalities of two *compute_subtreeMatch* are similar to that of algorithm 1. The *compute_subtreeMatch* in algorithm 2 considers n

as an only element in node's *LinkMatchList* and returns all new generated subtreeMatches, whereas the second *compute_subtreeMatch* uses *list_of_subtrees* as S_i of the child node which sends *list_of_subtrees*.

Algorithm 4 (remove a dependency link)

postiterate each pattern node except the root

    $n = modifier_of_dependencylink$

    if n is in *LinkMatchList*

        remove n from *LinkMatchList*

        for each *subtree* in *SubtreeMatchList*

            if $root_of_subtree = n$

                remove *subtree* from *SubtreeMatchList*

                $h = head_of_dependencylink$

        if h is not null

            send h to *parent_node*

Algorithm 5 (parent node receives h)

for each *subtree* in *SubtreeMatchList*

    if $root_of_subtree = h$

        remove *subtree* from *SubtreeMatchList*

        $h' = parent_of_subtree$

if h' is not null and node is not the root

    $h = h'$

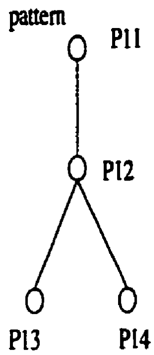
    send h to its parent

Let us use the example in Figure 5.4 to illustrate the algorithm. Suppose the linkMatches of the nodes of two rule patterns are:

```
P12 LinkMatchList:{ D3 }
```

Rule 1

pattern

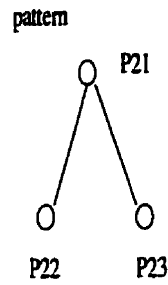


action

singleTransfer(0 0.1 0.1.1)

Rule 2

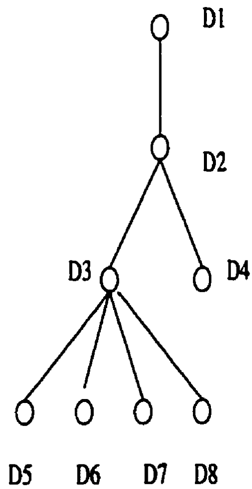
pattern



action

delete(0 0.1)

The dependency tree before the modification



The dependency tree after the modification

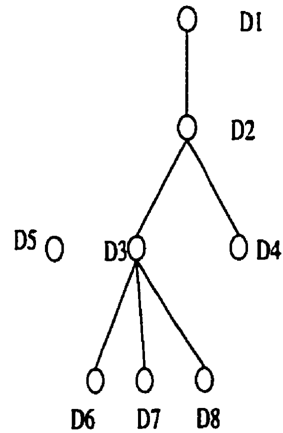


Figure 5.4: The example of the Tree Modifying Algorithm

```
P13 LinkMatchList:{ D5 }
P14 LinkMatchList:{ D6 }

P22 LinkMatchList:{ }
P23 LinkMatchList:{ D4 }
```

The pattern nodes of two rules are postiterated to initialize their *SubtreeMatchList*. After the initialization, the *SubtreeMatchList* of the pattern nodes becomes:

```
P13 SubtreeMatchList:{ (D5) }
P14 SubtreeMatchList:{ (D6) }
P12 SubtreeMatchList:{ (D3 (D5 D6)) }
P11 SubtreeMatchList:{ (D2 (D3 (D5 D6))) }

P21 SubtreeMatchList:{ }
P22 SubtreeMatchList:{ }
P23 SubtreeMatchList:{ (D4) }
```

If Rule 1 is applied first, it picks up the first element in its pattern root's *SubtreeMatchList* ($D2$ ($D3$ ($D5$ $D6$))) and modifies this subtree by two steps: remove $D5 \rightarrow D3$ and add $D5 \rightarrow D2$. While $D5 \rightarrow D3$ is being removed from the dependency tree, each non-root node in Rule 1 and Rule 2 checks if $D5$ is in its *LinkMatchList*, and if so, the node recomputes *LinkMatchList* and *SubtreeMatchList*. In this example, as $D5$ is the node $P13$'s linkMatch, $P13$ removes $D5$ from its *LinkMatchList* and ($D5$) from its *SubtreeMatchList*. In the meantime, $P13$ notifies its parent $P12$ by sending it a message which contains $D3$. $P12$ removes its subtreeMatch according to the receiving message. Then $P12$ continues to send $D2$ to $P11$ and $P11$ removes the element in $P11$'s *SubtreeMatchList*.

While $D5 \rightarrow D3$ is being added to the dependency tree, each non-root node of Rule 1 and Rule 2 checks if $D5$ is its new linkMatch. Suppose $D5$ is the $P22$'s linkMatch. $P22$'s two memories become:

```
P22 LinkMatchList   :{ D5 }
P22 SubtreeMatchList:{ (D5) }
```

$P22$ also sends its new subtreeMatch ($D5$) to $P21$ and $P21$ generates new subtreeMatch ($D2$ ($D5$ $D4$)). Rule 2 picks up this subtree and removes $D5 \rightarrow D2$ by following the same process. Then the *SubtreeMatchList* of two rule's pattern roots are empty and the modification process stops. Finally, the memories of the pattern nodes are:

```
P13 LinkMatchList   :{ }
P13 SubtreeMatchList:{ }
P14 LinkMatchList   :{ D6 }
P14 SubtreeMatchList:{ (D6) }
P12 LinkMatchList   :{ D3 }
P12 SubtreeMatchList:{ }
P11 SubtreeMatchList:{ }

P22 SubtreeMatchList:{ }
P22 SubtreeMatchList:{ }
P23 LinkMatchList   :{ (D4) }
P23 SubtreeMatchList:{ (D4) }
P21 SubtreeMatchList:{ }
```

5.3 Class ModificationModule

In the system, the modification module, rule, pattern and pattern nodes are implemented as classes. The object model of these classes is in Figure 5.5. The class

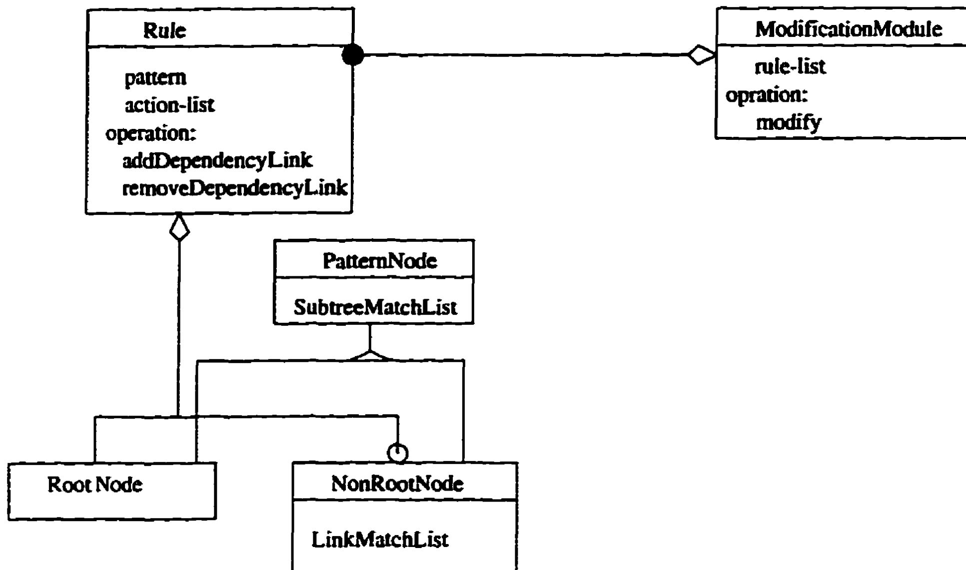


Figure 5.5: The Object Model which uses Rumgaugh OMT notation

RootNode and NonRootNode are derived from the base class PatternNode. The class NonRootNode has two attributes: LinkMatchList and SubtreeMatchList. The **LinkMatchList** is a list of the references of linkMatches and **SubtreeMatchList** includes a list of the references of subtreeMatches. The reference is used to refer to the corresponding linkMatch or subtreeMatch. In the class RootNode, the attribute **SubtreeMatchList** is a list of the references of the subparsetrees which match the whole pattern. The algorithms in the last section are implemented in the operation **buildInitialMatches, addDependencyLink** and **removeDependencyLink** of the class Rule. The Tree Modifying Algorithm is implemented in the class ModificationModule operation **modify**:

```
modify(DependencyTree tree)
{
   for each rule in rule-list
       rule.buildInitialMatches(tree)

   for each rule in which the pattern root has non-empty SubtreeMatchList {
```

```
        while(rule.pickSubparsetree) {
            action-sequence=translateAction;
            /* translate to a sequence of removing or adding a dependency link */
            for each add or remove in action_sequence{
                tree.addDependencyLink or tree.removeDependencyLink
                for each rule in rule-list
                    rule.addDependecyLink or rule.removeDependecyLink
            }
        }
    }
}
```

The evaluation system is coded in C++ on a Unix platform. To simplify the implementation of the Tree Modifying Algorithm, we make one assumption: if several sibling nodes of a dependency tree are the linkMatches of the same pattern node, only one of those sibling nodes is considered as the linkMatch, and others are ignored. the assumption largely reduces the complexity of the implementation and the system is still able to fulfill all modification requirements we have encountered.

5.4 Experimental Results

An experiment of the evaluation system is performed to evaluate the PRINCIPAR [14]. The treebank used for the experiment is from the SUSANNE Corpus Rel 3.0.

The SUSANNE Corpus was developed by University of Sussex. Release 3.0 was completed in 1994. The SUSANNE Corpus comprises an approximately 130,000-word subset of the Brown Corpus of American English, annotated in accordance with the SUSANNE scheme.

The SUSANNE corpus has 64 files which are classified into 4 types:

- A: press reportage
- G: belles letters, biography, memoirs
- J: learned (mainly scientific and technical) writing
- N: adventure and Western fiction

In the experiment, we pick up two files from each type and each file has about 2000 words. Both SUSANNE parses and PRINCIPAR parses are transformed into dependency trees before evaluation. The result of the experiment is given in Table 5.1.

file	# of words	error rate	recall	precision
A01	2195	19.91	77.21	82.94
A02	2203	20.74	76.96	81.50
G01	2221	25.62	71.12	82.17
G02	2266	23.35	73.59	81.99
J01	2200	23.27	75.29	79.04
J02	2089	23.98	75.09	78.40
N01	2287	23.61	70.98	81.44
N02	2206	23.75	71.57	82.11

Table 5.1: The experimental results

The evaluation is performed in the general mode. The relation types of dependency links and the categories of words are ignored and two dependency links are considered to be equivalent as long as they have the same word as head.

The eight files are from four different domains. According to the experiment, the scores of the eight files are very close. PRINCIPAR performs quite consistently across four different domains.

Chapter 6

Evaluation for Ambiguous Sentences

In Chapter 3, the evaluation method works only if its natural language sentence has only one corresponding parse tree. However, some sentences may be structurally ambiguous. Therefore, there can be more than one parse tree associated with such a sentence. Consider the ambiguous sentence: "Flying planes could be dangerous"; Figure 6.1 shows its two possible parses.

In this chapter, several issues in the evaluation of ambiguous sentences are discussed.

6.1 Representation of Multiple Parse Trees

Because the answer and the key of a sentence may consist of more than one parse tree, the previous dependency tree representation has to be extended to accommodate multiple parse trees. The modified representation hierarchy is shown as

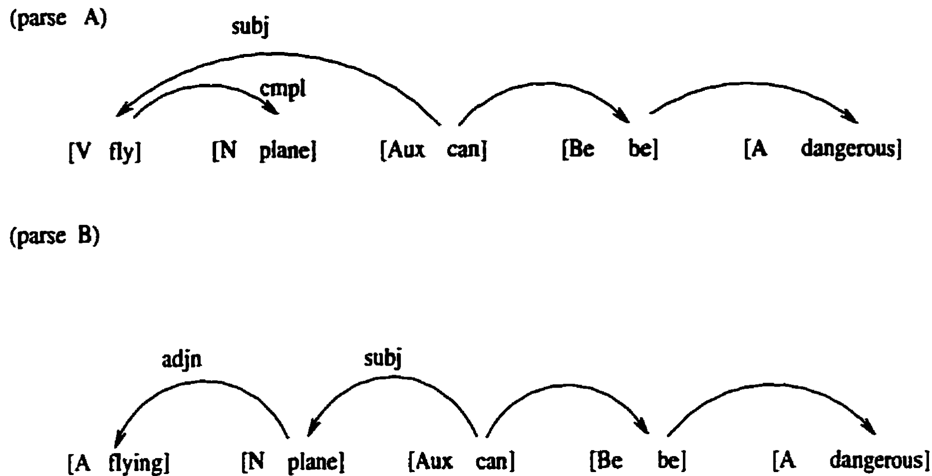


Figure 6.1: Two possible parses for "flying planes could be dangerous"

follows:

- dep-tree: (word-slot, word-slot,..., word-slot)
- word-slot: (word-token, wordsense, wordsense, ...)
- wordsense: (root, category, dependency-link, dependency-link, ...)
- root: the root form of the word
- category: the part of the speech tag of the word
- dependency-link: (head category position [relation type])
- dependency-link: `*` | `?`
- position: `<` `or` `>` `or` `<<` `or` `>>` `or ...`
- relation-type: `specifier or complement or adjunct or subject or ...`

In the above hierarchy, a word-slot includes a set of wordsenses, and a wordsense has a set of dependency links; whereas in the earlier representation, a word-slot has only one wordsense, and a wordsense has only one dependency link. Therefore, all possible parse trees of a sentence can be packed into a single representation. This representation can be called a parse forest which is equivalent to the share parse forest used by PRINCIPAR [14]. The following is the parse forest sample for: "flying planes could be dangerous":

```
(
 (flying (fly V (< can Aux subj))
         (flying A (< planes N adjn)))
 (planes (plane N (< flying V cmpl)
                  (< could Aux subj)))
 (could  (can Aux *))
 (be     (be Be (> could Aux)))
 (dangerous (dangerous A (> be Be pred)))
)
```

6.2 Evaluation Metrics

After the answer and key are represented by parse forests, it is straightforward to extend the recall-precision metrics accordingly.

Based on the general criterion of recall-precision , the formulas of precision and recall for multiple parses are presented as follows:

$$precision = \frac{\sum_{all\ word\ slots} size(intersection(K, A))}{\sum_{all\ A}(size(A))}$$

$$recall = \frac{\sum_{all\ word\ slots} size(intersection(K, A))}{\sum_{all\ K}(size(K))}$$

word	K	A	intersection(A, K)
flying	(< plane N adjn) (< can Aux subj)	(< plane N adjn)	(< plane N adjn)
planes	(< flying V cmpl) (< could Aux subj)	(< could Aux subj) (< be Be subj)	(< could Aux subj)
could	*	* (< be Be)	*
be	(> could Aux)	(> could Aux) *	(> could Aux)
dangerous	(> be Be pred)	(> be Be pred)	(> be Be pred)
SUM	7	8	5

Recall = 5/7 = 71.4% Precision = 5/8 = 62.5%

Table 6.1: Hamming Distance, Recall and Precision

For example, using the parse forest in the last section as the key and the following one as the answer, the evaluation metrics can be computed as shown in Table 6.1.

```
(
 (flying (flying A (< planes N adjn)))
 (planes (plane N  (< could Aux subj)
                   (< be Be subj)))
 (could  (can Aux  *
                   (< be Be)))
 (be     (be Be (> could Aux)
               *))
 (dangerous (dangerous A (> be Be pred)))
)
```

6.3 The Problem of Modifying Parse Forests

In contrast to the evaluation metrics, the modification tool in a parse forest is difficult to implement. A rule modifies a portion of one dependency tree, if it matches the given condition. In a parse forest, there is no way to determine if a set of dependency links belong to one single parse tree. Therefore, a rule may change some dependency links which it does not intend to change.

For example, the sentence: "I saw a man with a dog and a cat" has two possible readings. One indicates that a person saw a man and a cat: another shows that a person saw a man and the man has a cat and a dog. Parse A and parse B in Figure 6.2 are associated with two readings respectively. The modification rule is used to build direct links between a verb and its complements, when a conjuncture node intervenes. While parse A is not affected by the rule, B is transformed to B' by the rule and parse forest D is derived by packing A and B'. On the other hand, C is the parse forest which packs A and B and the rule transforms C to C'. As reader may notice, C' is not equal to D. In the other words,

$$modify(pack(A, B)) \neq pack(modify(A), modify(B))$$

While D is the expected result, parse forest C' does not represent the meaning of the original two parse trees.

The modifying parse forest needs to be further studied. One possible solution is to use the tree structure to represent the parses of an ambiguous sentence for the modification and then pack them into a forest just before the evaluation.

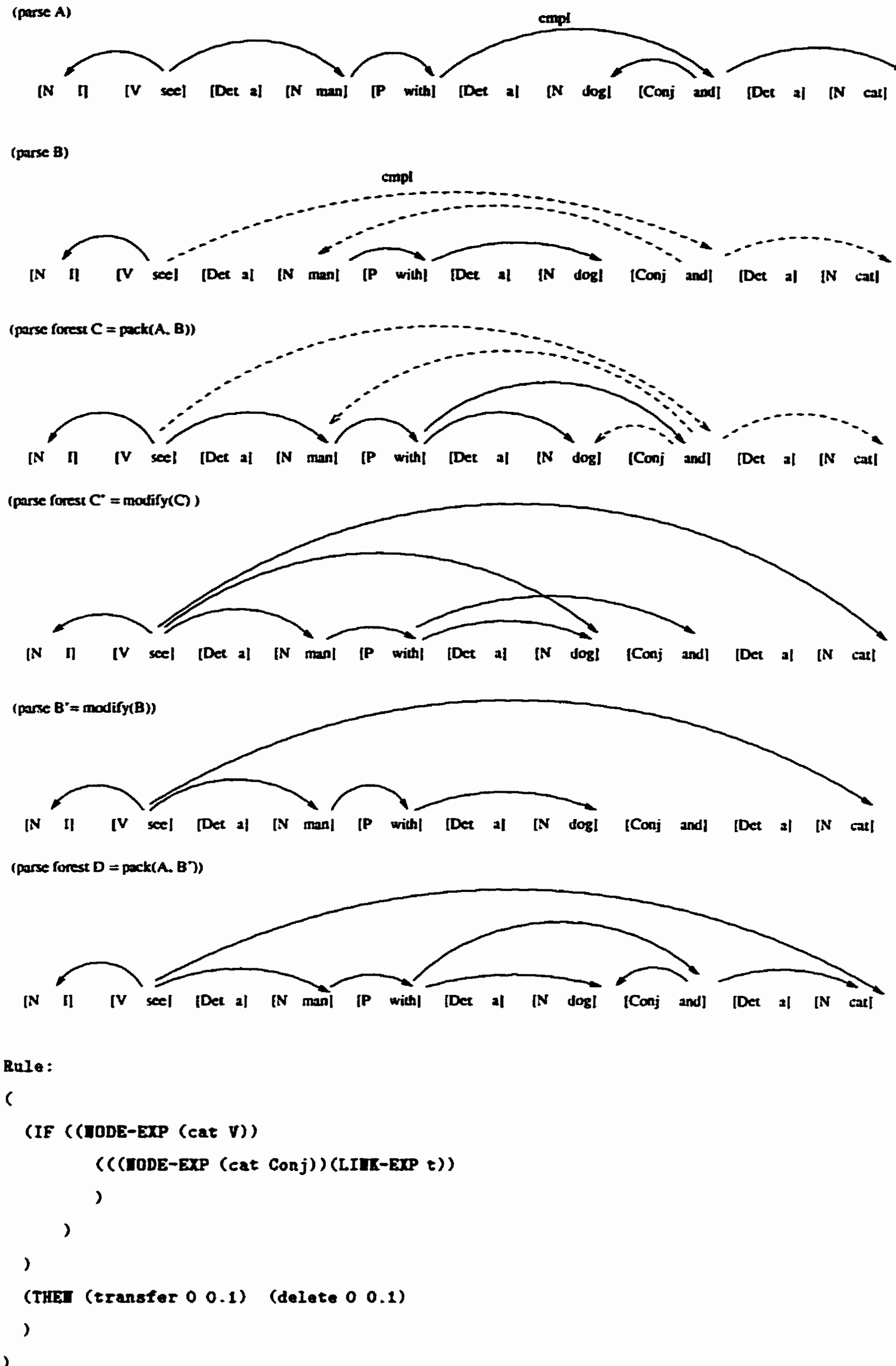


Figure 6.2: An example for modifying a parse forest. The subtrees match the condition are represented by dotted lines

Chapter 7

Conclusion

This chapter summarizes the thesis and suggests some future works.

7.1 Summary

This thesis studies the area of the dependency-based parser evaluation. Like other evaluation methods, the dependency-based evaluation uses treebanks as a standard. However, this method adopts dependency trees as formal syntactic representations of treebank parses and parser generated parses. The evaluation metrics are derived by comparing the dependency relations in treebank parses to parser generated parses. Besides the primary metric *error rate*, the other two metrics, *recall and precision*, are also introduced into the method, since recall and precision represent two important characteristics (completeness and accuracy) of the performance. These metrics can be extended to handle ambiguous sentences.

The modification tool presented in the thesis can transform dependency trees to

normalize different parsers according to a set of rules.

Furthermore, the experiment which is based on a corpus (16,000 words) demonstrates that the method is technically feasible in parser evaluation.

7.2 Future Work

As mentioned in the last chapter, future study of the modification tool needs to be performed in order to modify parse forests directly.

Another area of potential research is to assign weights to errors according to their severeness. Different errors have different impacts on the analysis of a particular sentence. It may be necessary for evaluation metrics to take this difference into account.

Appendix A

Sample parse trees from treebanks

A sample parse from Lancaster Treebank

```
[Fa If_CS
    [N you_PPY N]
    [V were_VBDR using_VVG
       [N a_AT1 shared_JJ folder_NN1 N]V]Fa]
[,_,
[V include_VVC
   [N the_AT following_JJ N]V]:_:
```

A sample parse from UPenn

```
(S (NP I)
```

```
(VP made
    (NP a list
          (PP of
              (SBAR (WHNP who)
                    (S (NP T)
                       could
                       (VP come))))))
```

A sample parse from Sussane

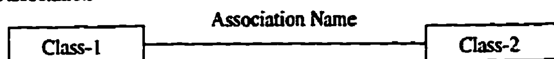
```
CSn     When          when          [S[Fa:t[Rq:t.Rq:t]
RR      sufficiently  sufficiently  [Np:s.
JJ      accurate      accurate      [JJ&.
CC      and           and           [JJ+.
JJ      complete      complete      .JJ+]JJ&]
NN2     measurements  measurement   .Np:s]
VBR     are           be            [Vab.Vab]
JJ      available     available     [J:e.J:e]Fa:t]
YC      +,            -             .
PPH1    it            it            [Ni:S.Ni:S]
VMo     will          will          [Vcb.
VB0     be            be            .Vcb]
JJ      possible      possible      [J:e.J:e]
TO      to            to            [Ti:s[Vi.
VV0v    set           set           .Vi]
```

```
NN2     limits  limit   [Np:o.Np:o]
II      on      on      [P:p.
AT      the     the     [Np.
JJ      thermal thermal [JJ&.
CC      and     and     [JJ+.
JJ      electrical      electrical      .JJ+]JJ&]
NN2     characteristics characteristic  .
IO      of      of      [Po.
AT      the     the     [Np.
NN1c    surface surface [NN1c&.
CC      and     and     [NN1c+.
NN1c    subsurface      sub<hyphen>surface      .NN1c+]NN1c&]
NN2     materials       material        .
IO      of      of      [Po.
NN1c    moon    moon    .Nns]Po]Np]Po]Np]P:p]Ti:s]S]
YF      +.      -       .O]
```

Appendix B

OMT notations

(a)Association



(b)Multiplicity of associations

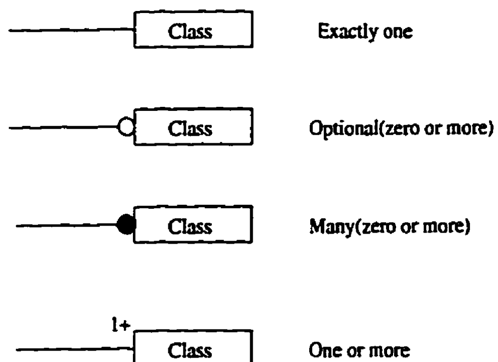


Figure B.1: Associations in OMT

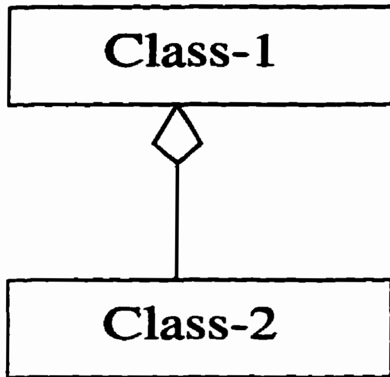


Figure B.2: Composition (aggregation) in OMT

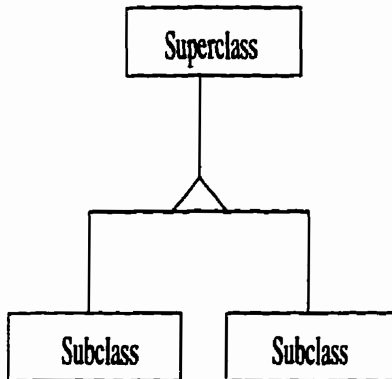


Figure B.3: Classification in OMT

Bibliography

[1] E. Black, S. Abney, D. Flickenger, C. Gdaniec, R. Grishman, P. Harrison, D. Hindle, R. Ingria, F. Jelinek, J. Klavans, M. Liberman, M. Marcus, S. Roukos, B Santorini, and T. Strzalkowski. A Procedure for Quantitatively Comparing the Syntactic Coverage of English Grammars. In *Proceedings of Speech and Natural Language Workshop*, pages 306–311. DARPA, February 1991.

[2] Ezra Black, John Lafferty, and Salim Roukos. Development and evaluation of a broad-coverage probabilistic grammar of English-language computer manuals. In *Proceedings of ACL-92*, pages 185–192, Newark, Delaware, 1992.

[3] Nancy Chinchor. MUC-5 Evaluation Metrics. In *Proceedings of the 5th Message Understanding Conference*, pages 69–78. ARPA, 1993.

[4] Michael Collins. A New Statistical Parser Based on Bigram Lexical Dependencies. In *Proceedings of ACL-96*. 1996.

[5] Christy Doran, Dania Egedi, Beth Ann Hockey, B. Srinivas, and Martin Zaidel. XTAG System - A Wide Coverage Grammar for English. In *Proceedings of COLING-94*, pages 922–928. Kyoto, Japan, 1994.

[6] Gerald Gazdar. *Natural Language Processing in LISP*. Addison-Wesley Publishing Company, 1989.

[7] Ian Graham. *Object Oriented Methods*. Addison-Wesley Publishing Company, 1994.

[8] Richard Hudson. Constituency and dependency. *Linguistics*, 18:179–198, 1980.

[9] Richard Hudson. *Word Grammar*. Basil Blackwell Ltd, 1984.

[10] Richard Hudson. *English Word Grammar*. Basil Blackwell Ltd, 1990.

[11] Joshi, A. Levy, and M. Takashi. Tree adjunct grammar. *Journal of Computer and System Sciences*, 1974.

[12] Dekang Lin. A Dependency-based Method for Evaluating Broad-coverage Parsers. In *Proceedings of IJCAI-95*.

[13] Dekang Lin. *Government-Binding Theory and Principle-based Parsing*, 1994.

[14] Dekang Lin. PRINCIPAR—An Efficient, Broad-coverage, Principle-based Parser. In *Proceedings of COLING-94*, pages 482–488. Kyoto, Japan, 1994.

[15] Dekang Lin. Principle-based parsing without overgeneration. In *Proceedings of ACL-93*, pages 112–120. Columbus, Ohio, 1994.

[16] Dekang Lin and Randy Goebel. Context-Free Grammar Parsing by Message Passing. In *Proceedings of the First Conference of the Pacific Association for Computational Linguistics*, pages 203–211, Vancouver, British Columbia, 1993.

[17] David M. Magerman. *Natural Language Parsing as Statistical Pattern Recognition*. PhD thesis, Stanford University, 1994.

[18] Igor A. Melcuk. *Dependency Syntax: Theory and Practice.* State University of New York Press, 1988.

[19] Berttran Meyer. *Object-Oriented Applications.* Prentice Hall, 1993.

[20] S Pertrick. Parsing. In *AI Encyclopedia*, pages 1099–1109. 1992.

[21] Geoffrey Sampson. *The Susanne Corpus*, 1994.

[22] Beatrice Santorini. *Bracketing Guidelines for the Penn Treebank Project*, 1991.

[23] Rajjan Shinghal. *Formal Concepts in Artificial Intelligence.* Chapman & Hall, 1992.