THE APPLICATION OF CONSTRAINT RULES TO DATA-DRIVEN PARSING

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2015

By
Sardar Jaf
School of Computer Science

Contents

Word Count: 51074

List of Tables

List of Figures

Abstract

THE APPLICATION OF CONSTRAINT RULES TO DATA-DRIVEN PARSING
Sardar Jaf
A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2015

The process of determining the structural relationships between words in both natural and machine languages is known as *parsing*. Parsers are used as core components in a number of Natural Language Processing (NLP) applications such as online tutoring applications, dialogue-based systems and textual entailment systems. They have been used widely in the development of machine languages.

In order to understand the way parsers work, we will investigate and describe a number of widely used parsing algorithms. These algorithms have been utilised in a range of different contexts such as dependency frameworks and phrase structure frameworks. We will investigate and describe some of the fundamental aspects of each of these frameworks, which can function in various ways including grammar-driven approaches and data-driven approaches. Grammar-driven approaches use a set of grammatical rules for determining the syntactic structures of sentences during parsing. Data-driven approaches use a set of parsed data to generate a parse model which is used for guiding the parser during the processing of new sentences. A number of state-of-the-art parsers have been developed that use such frameworks and approaches. We will briefly highlight some of these in this thesis. There are three specific important features that it is important to integrate into the development of parsers. These are *efficiency*, *accuracy*, and *robustness*. Efficiency is concerned with the use of as little time and computing resources as possible when processing natural language text. Accuracy involves maximising the correctness of the analyses that a parser produces. Robustness

is a measure of a parser's ability to cope with grammatically complex sentences and produce analyses of a large proportion of a set of sentences.

In this thesis, we present a parser that can efficiently, accurately, and robustly parse a set of natural language sentences. Additionally, the implementation of the parser presented here allows for some trading-off between different levels of parsing performance. For example, some NLP applications may emphasise efficiency/robustness over accuracy while some other NLP systems may require a greater focus on accuracy. In dialogue-based systems, it may be preferable to produce a correct grammatical analysis of a question, rather than incorrectly analysing the grammatical structure of a question or quickly producing a grammatically incorrect answer for a question. Alternatively, it may be desirable that document translation systems translate a document into a different language quickly but less accurately, rather than slowly but highly accurately, because users may be able to correct grammatically incorrect sentences manually if necessary. The parser presented here is based on data-driven approaches but we will allow for the application of constraint rules to it in order to improve its performance.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

dedication

We achieve what we dream to achieve.

In dedication to my mother and father; Xayria Abdul-Qadir Jaf and Fatih Mohammed Amin Jaf. Thank you for the love and devotion that gave me dreams to follow.

Acknowledgements

It has been one of the biggest challenges in my life to embark on a Ph.D. study in a reputable university. However, with the help of a number of people to whom I would like to express my gratitude, I have successfully completed it.

First of all, my deepest appreciation goes to my supervisor, Professor Allan Ramsay. I am very grateful to him for all the support and advice he continually offered me and he has been a great mentor for me. I am greatly in debt to him for his efforts and dedication throughout my Ph.D. study. He has been the best supervisor I have ever had in my entire student life. Without his insight, guidance and ideas I could not have overcome many of the hurdles and challenges that I have encountered during my study. It has been a great pleasure working with him.

No words are powerful enough to expressing my enormous gratitude to my parent whom have selflessly stood by me throughout my life. I am also very grateful to all the other members of my family for their support and their belief in me. I am deeply thankful to my fiancé (Lisa Ahmed) who has patiently waited for me to complete this study and has given me such great support. I will forever be indebted to each one of them.

I would also like to extend my gratitude to those who donated their precious time to discuss various parts of my thesis with me, especially Rebwar Kemal, Maytham Alabbas, Majed Alsabaan, Joakim Nivre and Johan Hall. I would also like to thank the academic support staff of the school of computer science at the University of Manchester.

I would like to express my appreciation to the Qatar National Research Fund for their financial support for my Ph.D. study. Without their contribution I would not have had the satisfaction of completing this thesis.

Despite all the hard work I have had to do throughout my Ph.D. study I have enjoyed my time as a Ph.D. student. I am grateful to everyone who has been part of my life during my study.

List of Abbreviations

The following is a list of various abbreviations used throughout the thesis.

Abbr.	**Full form**
1st	First Person
2nd	Second Person
3rd	Third Person
ABBREV	Abbreviation
ADJ	Adjective
AFP	Agence France Presse
AVM	Attribute-value Matrices
BNC	British National Corpus
C-Structure	Constituent Structure
CATiB	Columbia Arabic Treebank
CFG	Context Free Grammar
COMP	Complement
CONJ	Conjunction
COP	Copula Verb
CPOS	Coarse-grained Part-of-speech
CVG	Compositional Vector Grammar
CoNLL	Conference on Natural Language Learning
DATB	Dependency Arabic Treebank
DEF	Definite
DEPREL	Dependency Relations
DET	Determiner

Continued on next page

DF	Dependency Framework
DNDParser	Deterministic and Non-deterministic Parser
F-Structure	Functional Structure
FOM	Figure of Merit
FUT	Future
HMM	Hidden Markov Model
HPT	Head Percolation Table
ICONJ	Sentence-initial Conjunction
IV	Intransitive Verb
LALR	Look Ahead Left-to-Right
LAS	Labelled Attachment Score
LA	Labelled Attachment
LA	Left-ARC
LDC	Linguistic Data Consortium
LFG	Lexical Functional Grammar
LIBSVM	Library for Support Vector Machines
LOB	The Lancaster/Oslo-Bergen Corpus
LR	Left-to-Right
MADA	MorphologicalAanalysis and Disambiguation for Arabic
MOD	Modifier
MSA	Modern Standard Arabic
MSTParser	Maximum Spanning Tree Parser
MTT	Meaning Text Theory
MaltParser	Modelling and Algorithms for Language Technology Parser
MaxEnt	Maximum Entropy Inspired Parser
NLP	Natural Language Processing
NOM	Nominal
NP	Noun Phrase
NUM	Number
N	Noun
OBJ	Object
PADT	Prague Arabic Dependency Treebank
PATB	Penn Arabic Treebank

Continued on next page

PCFG	Probabilistic Context-free Grammar
PHONE	Phoneme
PNX	Punctuation
POS+LOC	Part-of-speech tag and Location
POS	Part-of-Speech
PP	Prepositional Phrase
PRED	Predicate
PREP	Preposition
PRON	Pronoun
PROP	Proper Noun
PRT	Particles
PSF	Phrase Structure Framework
PV	Perfect Verb
Pro-drop	Pronoun-dropping
RASP	Robust Accurate Statistical Parser
RA	Right-ARC
RNN	Recursive Neural Network
SBJ	Subject
SG	Singular
SVM	Support Vector Machine
SVO	Subject Verb Object
SV	Sentential Verb
S	Sentence
TV	Transitive Verb
TiMBL	Tilburg Memory-Based Learner
ULS	Unlabelled Attachment Score
VOS	Verb Object Subject
VP	Verb Phrase
VRB-PASS	Verb Passive
VSO	Verb Subject Object
V	Verb
WEAKA	Waikato Environment for Knowledge Analysis
WG	Word Gramar

Chapter 1

Introduction

1.1 Research overview

Parsing is a term used for the processing of human languages to determine the structural relations of words in sentences according to the grammatical rules of a language [Aho and Ullman, 1972, p. 63]. It is a core component in many Natural Language Processing (NLP) applications [Socher et al., 2013], such as machine translation, online tutoring applications, dialogue-based systems and textual entailment systems.

In this thesis, we will investigate a number of parsing algorithms, such as top-down parsing, bottom-up parsing, and chart parsing. We will also investigate different parsing frameworks, namely dependency framework and phrase structure framework. Parsers that are based on any of these frameworks process natural language sentences using one of two approaches:

1. grammar-driven approaches, where a set of grammatical rules is used for determining the syntactic structures of sentences during parsing.

2. data-driven approaches, where a set of parsed data is used for parser training and a parse model is generated which is used for processing new sentences.

We will briefly describe grammar-driven and data-driven approaches to parsing and highlight features of some of the state-of-the-art parsers.

The parser that we present in this thesis can efficiently, accurately, and robustly parse a set of natural language sentences (see Chapter 5). That is, it consumes as little time and computing resources as possible when processing natural language text (*efficiency*), it maximises the correctness of the analyses it produces (*accuracy*), and it can cope with grammatically complex sentences and produce analyses of a large proportion of a set of sentences (*robustness*).

NLP applications that use a parser as a subcomponent for processing text may trade-off certain aspects of parsing performance. Some NLP applications may require efficiency/robustness to be emphasised more than accuracy, while other applications may require more efficiency/robustness. For example, it may be important for dialogue-based systems to have a correct grammatical analysis of a question or produce a grammatically correct answer, rather than quickly but incorrectly analysing the grammatical structure of a question or quickly producing a grammatically incorrect answer for a question. It may, however, be desirable that document translation systems translate a document into a different language quickly but less accurately rather than slowly but highly accurately, because users can correct grammatically incorrect sentences manually if necessary. In this study, we present a novel methodology for integrating a set of constraint rules into a data-driven parser and show that we can trade-off effectively between accuracy and efficiency.

In this chapter, we will provide an overview of our research on processing natural language syntax. Moreover, we will explain the main challenges for parsing natural languages, and we will highlight our goals for conducting this research. We will also shed light on the main contributions of our work. Finally, we will conclude by outlining the main sections of this thesis.

1.2 An overview of the challenges in parsing natural languages

Natural languages contain ambiguities that make the task of developing a parser difficult and challenging [Farghaly and Shaalan, 2009, Holes, 2004, Collins, 2003].

In many natural languages, especially those such as Arabic with morphologically complex structures, a word may have multiple potential syntactic roles in a sentence. For example, a word can be a verb or a noun. Additionally, a verb can be intransitive (requiring no objects), monotransitive (requiring at least one object), or ditransitive (requiring two objects: a direct object and an indirect object). Parsers need to identify all possible interpretations for an ambiguous word/sentence so that they can produce correct analyses for them, which means that parsers must produce multiple analyses for an ambiguous word/sentence.

Producing multiple analyses for ambiguous words/sentences can have a knock-on effect on a parser's performance (efficiency, accuracy, and robustness). Also, conducting multiple analyses of a single word/sentence requires more time and computing resources than one analysis. It is more likely that a parser will select an incorrect analysis as its final answer when presented with multiple analyses of a single word/sentence. Finally, exploring multiple avenues for parsing ambiguous words or sentences can result in the production of several partial analyses that may exhaust available time and memory and lead to failure to deliver a final analysis. Example (1) shows that the word *saw* could be interpreted as a noun, referring to the name of a tool for cutting a tree, or as a verb indicating seeing a tree in a park. To produce an accurate interpretation, parsers have to explore and produce different analyses based on the role of *saw* in the sentence.

(1) I saw a tree in the park with a telescope.

Additionally, word order freedom in natural languages, where individual parts of a sentence need not necessarily be placed in a firmly given sequence, has a significant effect on parsing performance. Re-ordering words in sentences may not only alter the original meaning of sentences, but may also make it difficult for a parser to identify subjects or objects of sentences. For example, it is hard for parsers to determine the subject or the object of the Arabic sentence in 2(a) and 2(b). In 2(a) it is clear that الجندي *aljndy* "the soldier" is the subject of the verb and that الإرهابي *al-'irhAby* "the terrorist" is the object of the verb. However, reordering the words of the same sentence as in 2(b), makes it hard to determine whether الجندي *aljndy* "the soldier" is the subject of the verb as in قاتل الجندي الإرهابي *qAtl aljndy al-'irhAby* "the soldier fought the terrorist" or الإرهابي *al-'irhAby* "the terrorist" is the subject as in قاتل الجندي الإرهابي *qAtl aljndy al-'irhAby* "the killer of the soldier is the terrorist".

(2)

(a) الجندي قاتل الإرهابي *aljndy qAtl al-'irhAby* "the soldier fought the terrorist"

(b) قاتل الجندي الإرهابي *qAtl aljndy al-'irhAby* "the killer of the soldier is the terrorist"

Furthermore, in some morphologically rich languages, verbs can indicate the gender, number and the person of the subject of a sentence. In such languages, it is possible to completely omit subjects from sentences. For example, the subject in the Arabic sentence in (3) is completely removed because the verb أكلت *'kalat* "ate" is rich enough to indicate that the subject is feminine, singular, and second person. However, the omission of sentences' subjects can make it difficult for parsers to determine whether the subject is actually being omitted or whether the verb has one object (monotransitive). There are at least two analyses for the sentence in (3), (i) the sentence has no objects and the word الدجاجة *AldajAjT* "the chicken" is the subject of the verb أكلت *'kalat* "ate", and (ii) the subject of the sentence is "she" but it is removed from the sentence because the verb أكلت *'kalat* "ate" can recover it, and the object is الدجاجة *AldajAjT* "the chicken".

In Chapter 4, we will explain in more detail a number of natural language ambiguities that affect parsers.

(3) أكلت الدجاجة *'kalat Al dajAjT* "the chicken ate/She ate the chicken"

1.3 Research goals

Our goal in this study is to implement a parser for processing natural languages, with a particular focus on Arabic because it is more ambiguous than some other languages [Farghaly and Shaalan, 2009, Chalabi, 2004a, Holes, 2004, Daimi, 2001, Fehri, 1993], such as English, and hence can be particularly challenging. The main objectives of this study are as follows:

- To investigate current parsing algorithms.
- To explore different frameworks and approaches to parsing natural languages.
- To identify different ways of extracting constraint rules from treebanks.
- To integrate a set of constraint rules into a data-driven parser.
- To be able to trade-off between different features of parsing (efficiency, accuracy and robustness).

1.4 Research contribution

Parsing is a core component in a number of NLP applications. An efficient, robust, and accurate parser may greatly contribute to those applications. There are a number of parsing algorithms, frameworks and approaches to parsing, each one of which has certain limitations. However, each offers different benefits.

On the one hand, parsers that are trained on a treebank using machine learning algorithms can be efficient and robust. These parsers are categorised as data-driven parsers [McDonald et al., 2006, Nivre, 2006]. They generally produce some analyses for any input sentences, based on what they have learned from the set of data they were trained on. Data-driven parsers do not implement linguistic grammatical rules for analysing the grammaticality of the sentences they process. Hence they may produce analyses that do not correspond to the structures defined by standard linguistic grammar.

On the other hand, parsers that use grammatical rules for analysing sentences are categorised as grammar-driven parsers. These parsers often produce highly accurate results but may have problems with efficiency and robustness for three reasons: (i) the grammatical rules used in these parsers may not cover the whole language in question, (ii) some sentences might fall outside norms of the language in question such as incurerectly spelled words, and (iii) they have to explore all possible analyses for ambiguous words/sentences, which can make them very slow.

The main contributions of the research are:

C.1 We present an approach for directly integrating a set of constraint rules into a data-driven parser in order to improve the performance of a data-driven parser (see Section 5.2.2 for more details).

C.2 We present a technique for easily activating/deactivating the application of constraint rules to a data-driven parser, which will allow us to easily trade-off between accuracy and speed (see Section 6.5 for more details).

C.3 We present a method for extracting different sets of constraint rules from a dependency treebank, which is an easy and quick way for constructing rules that can be used as linguistic grammatical rules (see Section 6.5.2 for more details).

C.4 We present a new approach to non-projective parsing with a data-driven parser (see Section 5.2.1 for more details).

1.5 Research methodology

The methods of parsing human languages examined in this study have three main elements:

1. Parsing algorithms: A combination of two widely used parsing algorithms will be used to construct the parser presented in this thesis, which is a shift-reduce parsing algorithm with dynamic programming.

2. Constraint rules: We will integrate a set of constraint rules extracted from a dependency treebank into a data-driven parser to ensure that the analyses produced by our parser obey the specified constraints.

3. Machine learning algorithm: This is used for generating a parse model, which guides the parser to its next parsing action to improve the parser efficiency (i.e., to consume as little time and computing resources as possible) and also to assist the parser in producing analyses based on its learning from a set of data.

1.6 Thesis outline

In this introductory chapter, we have presented an overview of the research problem and the main challenges involved in parsing natural language texts. We have highlighted our aims and our research contributions, and have shed light on the research methodology that we are using for producing an efficient, robust, and accurate parser. The remainder of this thesis is organised as follows:

Chapter 2
Investigating Parsing Algorithms
Chapter 2 discusses the different natural language parsing algorithms in order to gain an understanding and identify the limitations of each of them.

Chapter 3
Natural Language Parsing Frameworks
Chapter 3 identifies and describes different frameworks and approaches to parsing natural languages and includes a brief description of a number of state-of-the-art parsers.

Chapter 4
The Challenges of Parsing Arabic
Chapter 4 focuses on highlighting the complexities of natural languages, using Arabic as an example because it is more ambiguous than other languages such as English. We will use various examples to demonstrate ways that natural language ambiguities affect parsing performance.

Chapter 5
Parser Development
Chapter 5 describes various aspects of the development of our parser, which includes the evaluation of a state-of-the-parser and the way we modify it.

Chapter 6
Parser Evaluation
Chapter 6 describes the evaluation of our data-driven parser. We also show the effect of applying a set of constraint rules to it.

Chapter 7
Conclusion, Contributions, and Future Work
Chapter 7 concludes with the final remarks of the thesis. The chapter ends by suggesting some directions for future work and research.

Chapter 2

Investigating Parsing Algorithms

Parsing algorithms use different strategies for enumerating the nodes and arcs of parse trees and they are characterised by the way they use these strategies. For example, in a top-down strategy, the parsing of each node is enumerated before any of its descendants, while, in a bottom-up parsing strategy, each node is enumerated after all its descendants.

In this chapter we will describe some of the most widely used parsing algorithms: top-down parsing, bottom-up parsing, left-corner parsing, chart parsing, and shift-reduce parsing. We will describe these algorithms in detail in the following sections. These algorithms can all be run either depth-first or breadth-first search strategy. The key issue here concerns what the algorithm should do if it hits a dead-end. In depth-first search, it will return to the most recent choice point and look for an alternative, whereas in breadth-first search it will return to the oldest choice point.

We will describe each parsing algorithm by using a single example sentence to explain and describe the way they process sentences to produce parse trees. It is possible to use any of these algorithms to produce parse trees by adopting two different frameworks, which are described in the next chapter. These are (i) a dependency framework, and (ii) a phrase structure framework. These frameworks can be used in different ways, in grammar-driven approaches where a set of grammatical rules is used for producing parse trees, and in data-driven approaches where a set of inferred rules extracted from a set of training data is used to produce parse trees. In this chapter we will follow a grammar-driven approach by using a set of defined grammatical rules for processing a natural language sentence[1].

[1]We use grammar-driven approaches with a phrase structure framework in the examples of this chapter because it is more convenient for describing the parsing algorithms.

Our goal in this chapter is to describe the parsing process of each algorithm in order to understand the way natural language ambiguities affect parsing performance. We will look at the traces of these algorithms in considerable detail to make it clear where backtracking occurs and what causes it to happen.

2.1 Top-down parsing algorithm

Top-down parsing algorithm break up the largest unit (a sentence) into smaller units (words and phrases) until a parse tree for a given input string is produced [Chapman, 1987]. In this algorithm, the process begins with the start symbol at the top of a parse tree and works downwards, applying and expanding rules forward until it reaches the terminal symbols (which can be either the actual words in the given sentence or their part-of-speech tags) [Aho and Ullman, 1972, p. 285]. The start symbol, which is the root of the parse tree, represents the sentence level, and the leaves which are at the bottom of the parse tree indicate the word level.

Top-down algorithms can process sentences according to a set of defined grammatical rules of a natural language [Thant et al., 2011]. Generally, top-down parsers perform three steps:

1. establishing the root of the parse tree and then expanding toward the leaves.
2. selecting a rule and finding an interpretation of an input string that can match it.
3. if the parser explores a rule and the analysis does not lead to a parse tree then the parser backtracks to a previous step and attempts an alternative interpretation.

We use the grammar rules in Figure 2.1 to parse the sentence *I know that man is happy* in order to illustrate the parsing process in the top-down algorithm, and to identify some of its limitations. We will be exploring the traces of this algorithm in considerable detail. The parsing steps in each parse tree are presented using numbers to indicate the parser's steps, and alternative analyses for each category are included below the categories.

The parsing algorithm begins to find the production of *S* by exploring the elements of *S* that are placed at the right-hand-side of the rule, as in the first line in Figure 2.1. Two non-terminal symbols[2], which indicate a noun-phrase (*NP*) and a verb phrase (*VP*)

[2]Non-terminal symbols are those symbols appearing on the left-hand-side of rules which have no direct links with the input strings, such as *S*, *NP*, and *VP* as shown in Figure 2.1.

S	→	NP	VP
NP	→	PRON	
NP	→	DET	N
VP	→	COP	ADJ
VP	→	TV	NP
VP	→	SV	S
PRON	→	I	
TV	→	know	
SV	→	know	
PRON	→	that	
DET	→	that	
TV	→	man	
N	→	man	
COP	→	is	
ADJ	→	happy	

Figure 2.1: Context-free grammar rules for parsing.

are used to create a partial parse tree, as shown in Figure 2.2(1). Next, the *NP* symbol is processed, because the strategy here is to parse from left to right and from top to bottom. The grammar rules indicate that an *NP* may consist of the symbol *PRON*, which the parser uses to create a new partial parse tree, as shown in Figure 2.2(2). At this stage the parser has an alternative option for producing the symbol NP, which is *DET* followed by *N*, as shown in the third rule in Figure 2.1, which is also under the *NP* node in the tree created by the parser as shown in Figure 2.2(2). This alternative analysis can be used by the parser during backtracking, in order to allow the parser to explore a different path for producing a complete parse tree for the sentence.

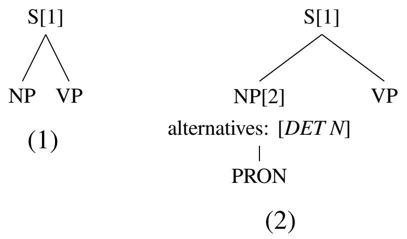


Figure 2.2: Top-down parse trees (1) and (2).

Next, the pre-terminal symbol[3] (*PRON*) is processed by using the terminal symbol[4] *I*, which is the first input string, as its production and a new partial parse tree is

[3]Pre-terminal symbols are intermediate symbols that appear between non-terminal symbols and terminal symbols, such as *N*, *V*, *DET* etc as in Figure 2.1.

[4]Terminal symbols are input strings in sentences, such as *I*, *know*, *man* etc. in Figure 2.1.

generated as in Figure 2.2(3). Here, we have followed the depth-first search strategy with top-down parsing by finding the daughter of the left branch of the tree before exploring the right branch. Alternatively, we could have followed the breadth-first search strategy by moving rightward to process the symbol *VP* after processing the symbol *NP* but before processing the symbol *PRON*. For example, following the breadth-first search strategy, the parser would have produced a tree similar to the one in Figure 2.2(4) instead of the one in Figure 2.2(3). Both depth-first or breadth-first strategies can be used when using a top-down algorithm. However, we will be using a depth-first search strategy in this algorithm.

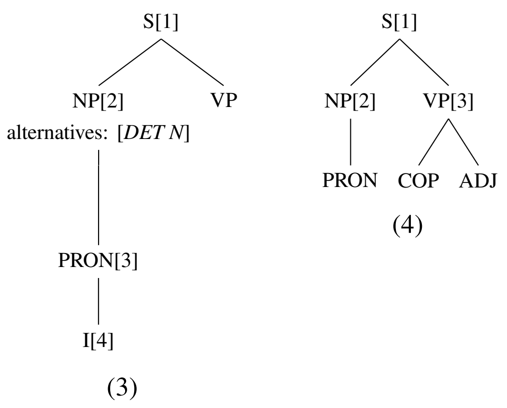


Figure 2.2: Top-down parse trees (3) and (4).

Once all of the daughters in the left branch are identified, then the parser moves rightwards to analyse the non-terminal symbol *VP* to generate a complete parse tree. Although the grammar rules indicate that the symbol *VP* has multiple interpretations, the pre-terminal symbols *COP* and *ADJ* are used as the production of the symbol *VP* because they are the first possible interpretation. *COP* and *ADJ* symbols are used for the production of the symbol *VP* and a new partial parse tree is generated, as shown in Figure 2.2(5). The parser then finds the next available word *know*, to be used as the production for the symbol *COP* from the grammar rules. However, the input string *know* is not defined as *COP*, so the parser backtracks to try the next interpretation of the symbol *VP*, which is *TV* followed by *NP*. Since we are following the depth-first search strategy, the parser backtracks to its last move, which was step 5, as in Figure 2.2(5), and it tries *VP* as *TV* followed by *NP*, as shown in Figure 2.2(6).

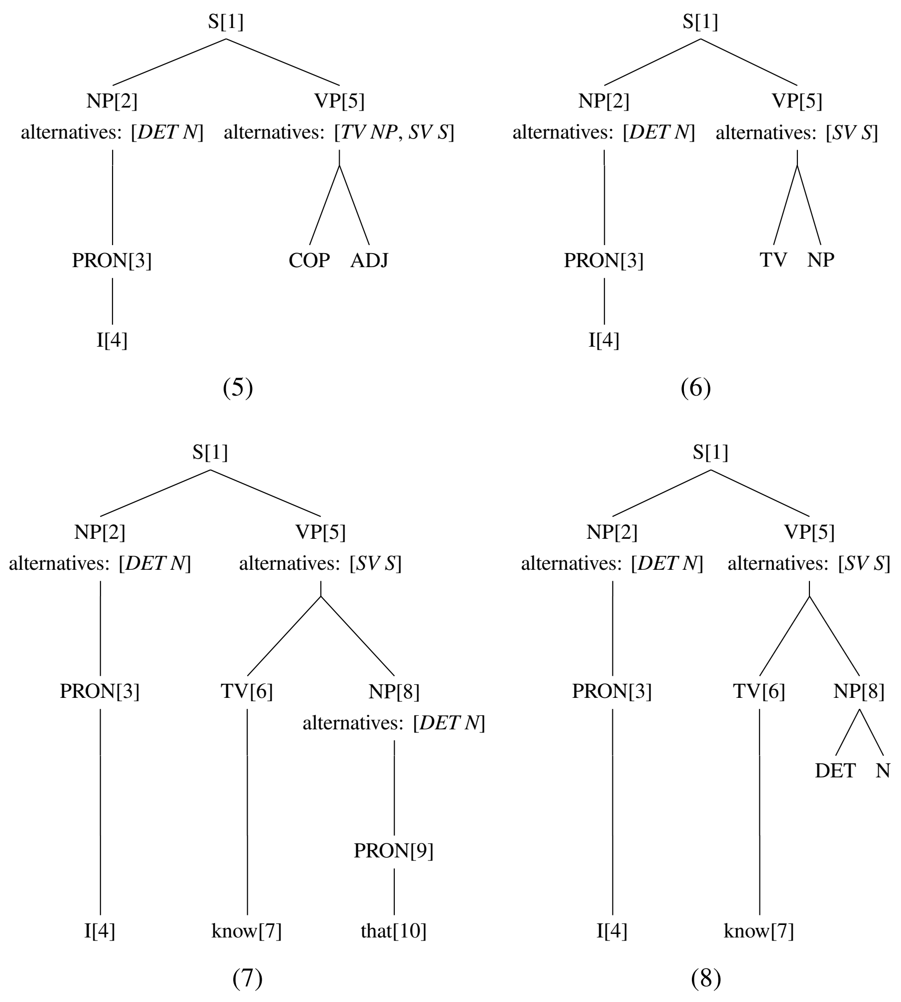


Figure 2.2: Top-down parse trees (5) and (8).

The algorithm then performs a number of parse steps and produces numerous intermediate parse trees[5] before generating a complete parse tree with the symbol *S* on the top as shown in Figure 2.2(7).

At this stage, the parser produces a full parse tree with the symbol *S* on the top. However, there are three more words (*man*, *is* and *happy*) remaining from the input strings for processing, which means that the sentence *I know that man is happy* is not fully parsed. Thus, the parser backtracks to find an alternative path that can lead to the production of a complete parse tree for the full sentence. The parser backtracks to its last step with an alternative analysis, which was step 8 of Figure 2.2(7), where the alternative analysis is to use *DET* followed by *N* as the production of *NP*, as in Figure 2.2(8).

[5]The full trace is given in Appendix A.

Next, the input strings *that* and *man* are processed and a full parse tree is again produced with the symbol *S* at the top, as in Figure 2.2(9).

Two more words (*is* and *happy*) now remain from the input string, indicating that the sentence (*I know that man is happy*) is not yet fully parsed. Backtracking occurs again. Since all of the analyses for the *NP* have been explored, the parser backtracks to step 5 from Figure 2.2(9) to try the alternative path for processing the non-terminal *VP* by using *SV* followed by *S*, as shown in Figure 2.2(10). This means that the parser goes back to start parsing the sentence from the input string *know*.

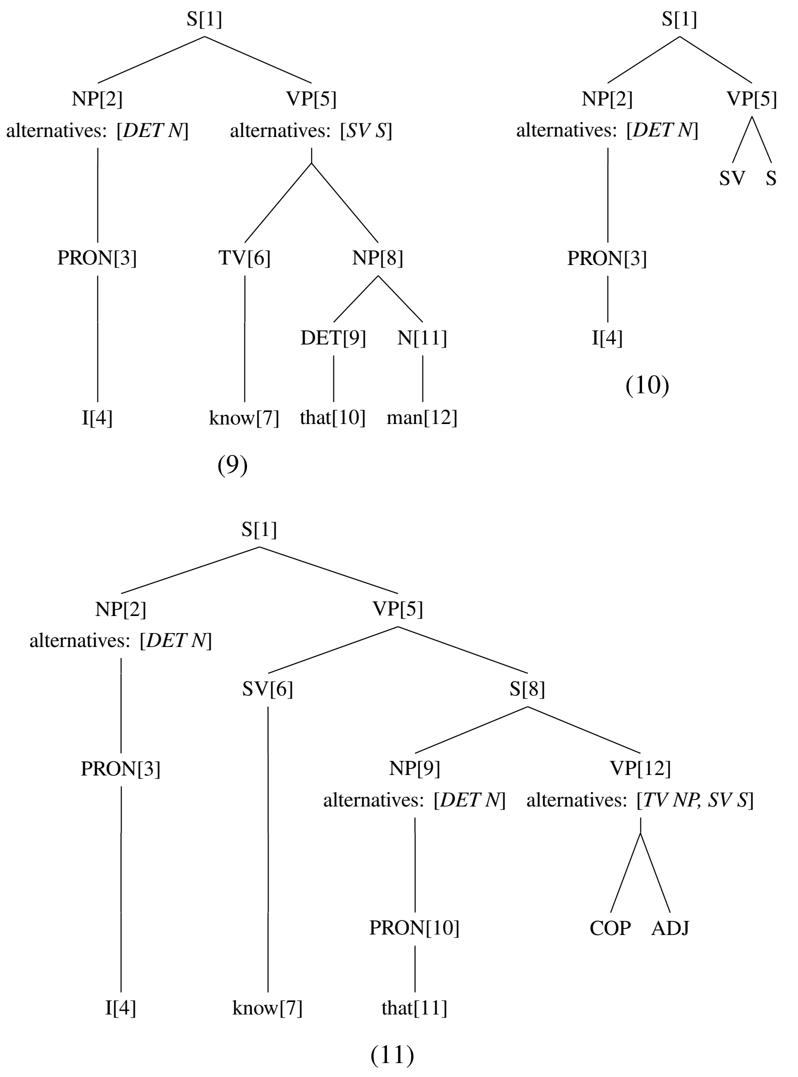


Figure 2.2: Top-down parse trees, from tree (9) to tree (11).

The algorithm attempts to analyse the sentence starting from the input word *know*, and it produces several parse trees as shown in Appendix A (from trees A.1(14) to tree A.1(17)), until it attempts to analyse the input word *man* where it tries to use *man* as the production for *COP*, which is not allowed by the grammatical rules of Figure 2.1. At this stage, the parser is in a state similar to that shown in Figure 2.2(11).

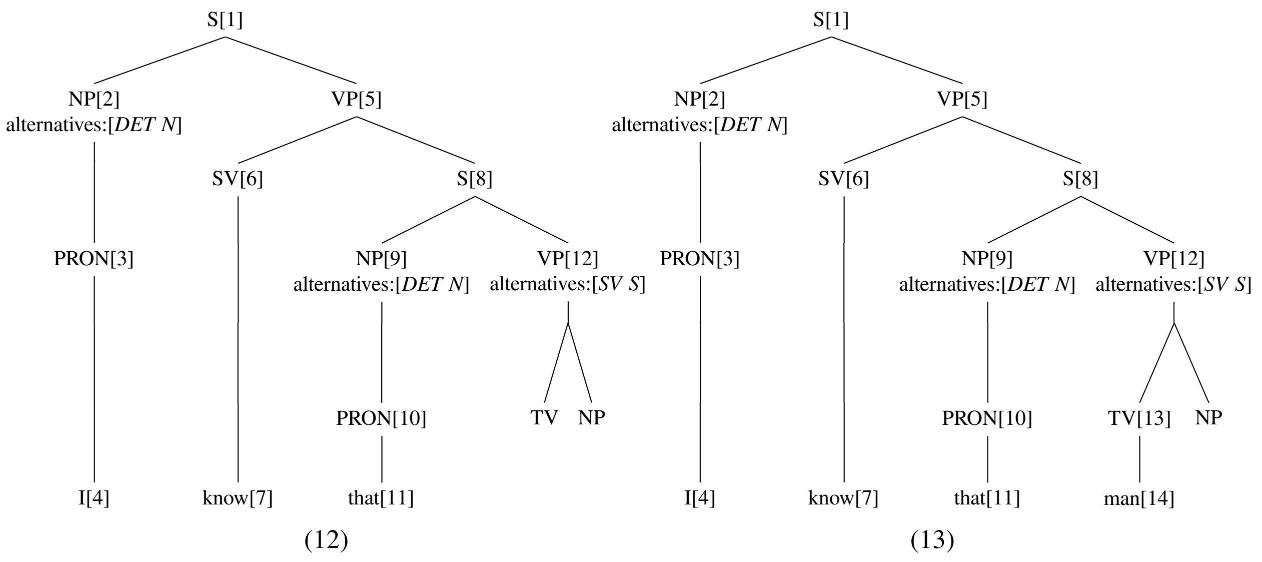


Figure 2.2: Top-down parse trees (12) and (13).

Since the next input string *man* cannot be used as the production of *COP*, the parser backtracks to step 12 of Figure 2.2(11) and uses *TV* followed by *NP* for the production of *VP*. A new partial parse tree is produced, as shown in Figure 2.2(12).

The algorithm then proceeds to use the input string *man* for the production of *TV* and then creates a new partial parse tree, as in Figure 2.2(13). However, the input string *is*, which is following the input string *man*, cannot be used for the production of *NP* so the parser backtracks to step 12 of Figure 2.2(13) and tries a different analysis of the *VP* by using *SV* followed by *S* for the production of *VP*. It then produces the new partial parse tree, as shown in Figure 2.2(14).

At this stage, the parser realises that the word *man* cannot be used as the production of *SV* and that it has exhausted all the possible alternatives for *VP*. Hence, it backtracks further to step 9 of Figure 2.2(14) to use the alternative path for the *NP*, which is using *DET* followed by *N* for the production of *NP*, and which then produces a new partial parse tree, as shown in Figure 2.2(15).

The word *that* is used as the production of *DET* and the word *man* is used as the production of *N*, which results in the production of a partial parse tree, as shown in Figure 2.2(16).

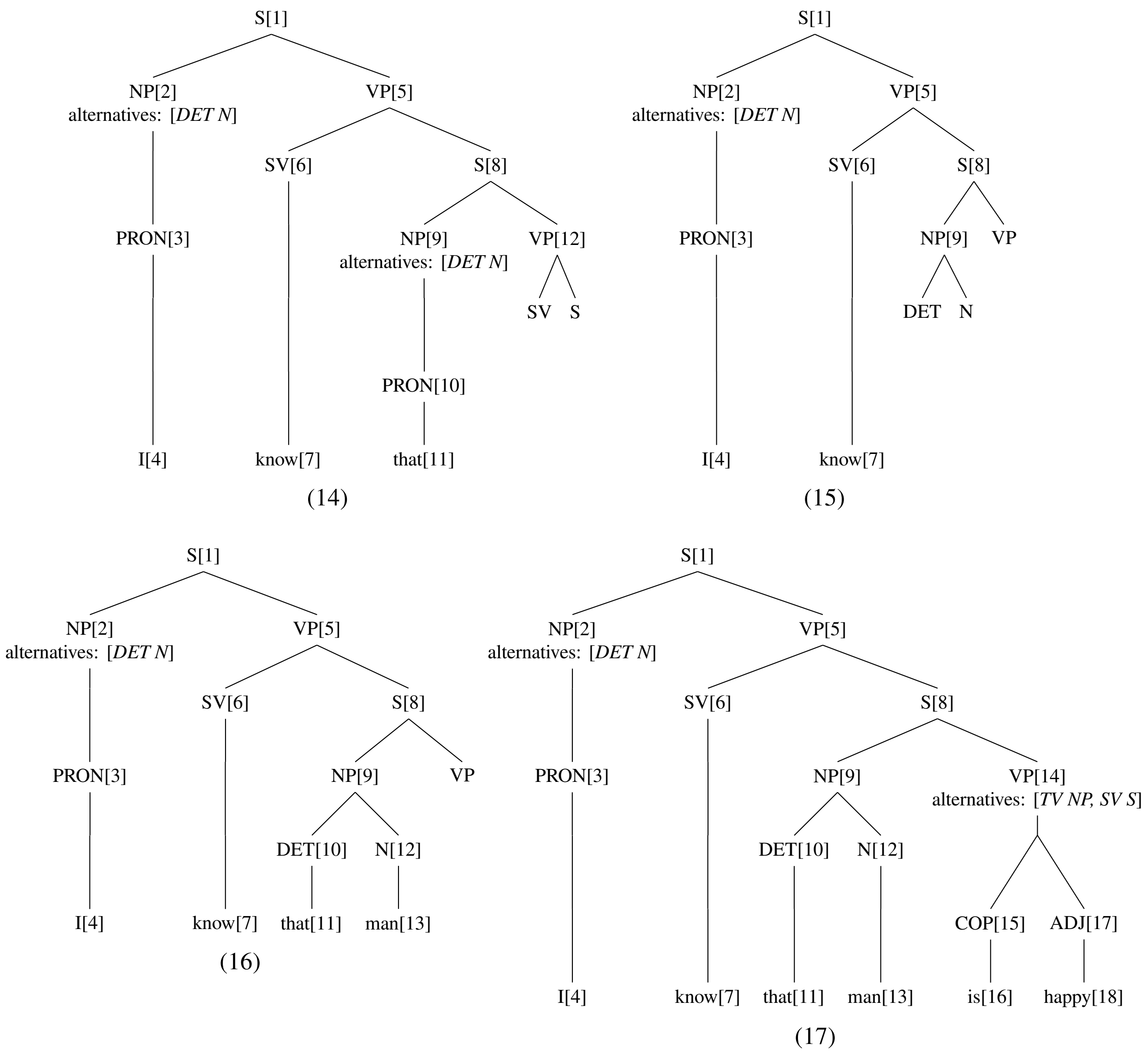


Figure 2.2: Top-down parse trees, from tree (14) to tree (17).

The non-terminal *VP*, where *COP* and *ADJ* are used for its production, is then processed while the input words *is* and *happy* are used for the production of the children of *VP*, resulting in the generation of a complete parse tree, as shown in Figure 2.2(17).

We can notice from processing the sentence *I know that man is happy* that top-down parsers are sometimes forced to retreat and choose alternative paths to re-analyse the same sentence. This is called backtracking. If the selected path leads to a dead end, the parser backtracks to a previous decision point, by moving backward and starting again using different paths until it either finds an appropriate production or runs out of choices.

We can conclude that one of the main reasons that backtracking occurs in top-down parsers is because parsers do not have any knowledge of the terminal symbols (words or input strings) when processing the pre-terminal symbols. Therefore, when it finds out that the available terminal symbol cannot be used as the production for the selected pre-terminal symbol, it backtracks to find alternative analyses that can be used for the production of the pre-terminal symbols in question, as we saw when the tree in Figure 2.2(11) was created, while the word *man* could not be used as the production of the pre-terminal symbol *COP*.

Backtracking is essential, because it allows the parser to explore alternative solutions when it encounters a failure. However, it may present problems [Chapman, 1987, p. 22]. It makes parsers inefficient because all the productions prior to the backtracking stage are lost, which may require the parser to repeat tasks that had already been carried out before backtracking such as the re-analysis of *that* and *man* from Figures 2.2(9) to 2.2(13) and 2.2(14) to 2.2(15), which means that the parser may consume more time and resources. We may also note that top-down parsers can require an exponential number of steps (with respect to the length of the sentence) to try all the alternative analyses of ambiguous sentences for producing a full parse tree for a complete sentence, which eventually requires exponential memory space.

In this section, we have described the traces of top-down parsing in considerable detail to identify the various situations where backtracking occurs. We will describe the traces of the other algorithms (bottom-up, left-corner etc.) in the same amount of detail in the following sections.

2.2 Bottom-up parsing algorithm

Bottom-up algorithm work the opposite way from top-down parsers. Parsing starts from the terminal symbols (input strings) which are, progressively, replaced by pre-terminals. Pre-terminals are then grouped together until they can be packaged into a single group that matches the root symbol, such as *S* for a given sentence [Mathews, 1998, p. 169] and [Chapman, 1987, p. 20]. In this algorithm, the goal is to use the terminal symbols as productions for pre-terminal symbols, then group pre-terminal symbols under non- terminal symbols in order to produce a single non-terminal symbol that represents the complete sentence.

In order to understand the process of analysing natural language sentences in bottom-up parsing, we will again parse the sentence *I know that man is happy*, which we used previously in Section 2.1, using the grammar rules in Figure 2.1. We will also follow a depth-first search strategy with this algorithm.

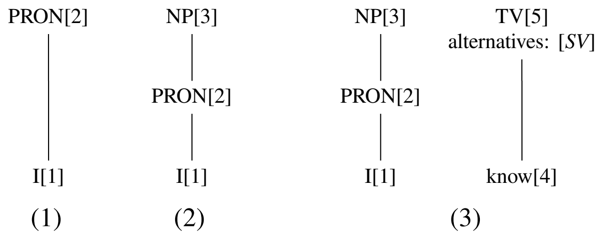


Figure 2.3: Bottom-up parse trees, from tree (1) to tree (3).

Initially, the algorithm starts processing from the terminal symbols (words or input strings). It will check for the lexical categories of the input words which are defined in the grammar rules in Figure 2.1. The parser processes the first word in the sentence, which is the word *I*. It replaces it with the pre-terminal symbol *PRON* and then it generates a partial tree as shown in Figure 2.3(1). The grammatical rules in Figure 2.1 permit the parser to produce the symbol *NP* from the symbol *PRON*. Hence, a new partial parse tree as in Figure 2.3(2) is produced. Since the symbol *NP* cannot produce any new symbols, then the next input string (*know*) is used for the production of the pre-terminal symbol *TV*[6] and a new partial parse tree is produced, as shown in Figure 2.3(3).

At this stage, the parser is presented with two partial parse trees, but, it cannot group them because there are no grammatical rules to permit the parser to group them.

[6]Parsers initially use the first available analysis for terminals that is produced from a set of grammatical rules. Hence, the parser uses *TV* for producing *know* because the first rule from the set of rules, as shown in Figure 2.1, indicates that *know* is the first possible production for *TV*.

The algorithm then proceeds to process the next input word *that* by replacing it with the pre-terminal symbol *PRON*, as in Figure 2.3(4). Since it is possible to produce the symbol *NP* from the symbol *PRON* using the grammar rules, a new partial parse tree is generated by the parser, as shown in Figure 2.3(5). Since the grammatical rules permit the parser to generate the symbol *VP* by combining the symbol *TV* with the symbol *NP*, a new tree is generated, as shown in Figure 2.3(6). Additionally, the grammatical rules permit the parser to generate the symbol *S* by combining the symbol *NP* with the symbol *VP*, as shown in Figure 2.3(7).

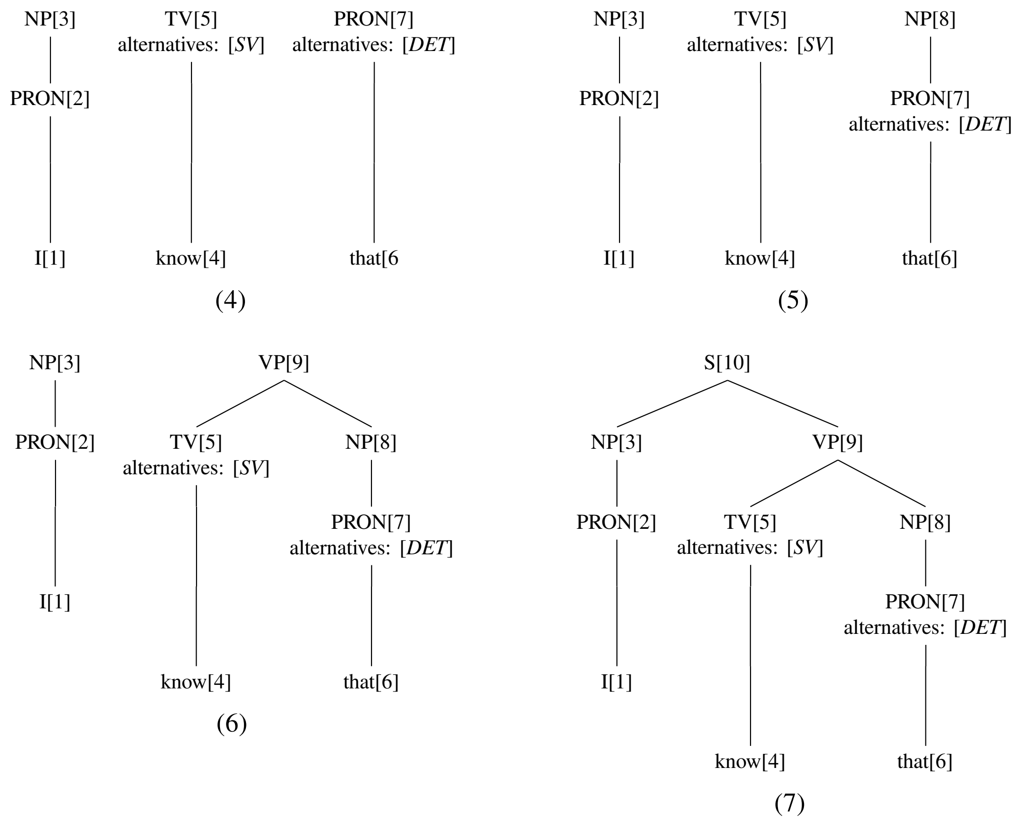


Figure 2.3: Bottom-up parse trees, from tree (4) to tree (7).

At this stage, the parser creates a full parse tree with the symbol *S* on the top. However, there are still some remaining words in the sentence that need to be processed. Hence, the parser backtracks to step 7 of Figure 2.3(7) where it has an alternative analysis of the word *that*, which processes it as *DET* and generates a new parse tree, as in Figure 2.3(8). The alternative analysis of the input word *that* does not lead to grouping the available pre-terminal symbols. Hence, the parser performs a number of steps and produces several trees before it can group some partial parse trees, as shown in Figure 2.3(9) (The generated partial parse trees between the tree in Figure 2.3(8) and Figure

2.3(9) are shown in Appendix A from tree A.2(9) to tree A.2(12)).

Since there are no more input words left for analysing and the parser cannot group any of the current non-terminal symbols, it backtracks to the last step where it had an alternative path to follow, which is step 9 of Figure 2.3(9), where the input word *man* is re-analysed as *N*, as shown in Figure 2.3(10). The parser then groups a number of partial parse trees (as shown in Appendix A from tree A.2(13) to tree A.2(15)) until it produces a full parse tree, as shown in Figure 2.3(11)

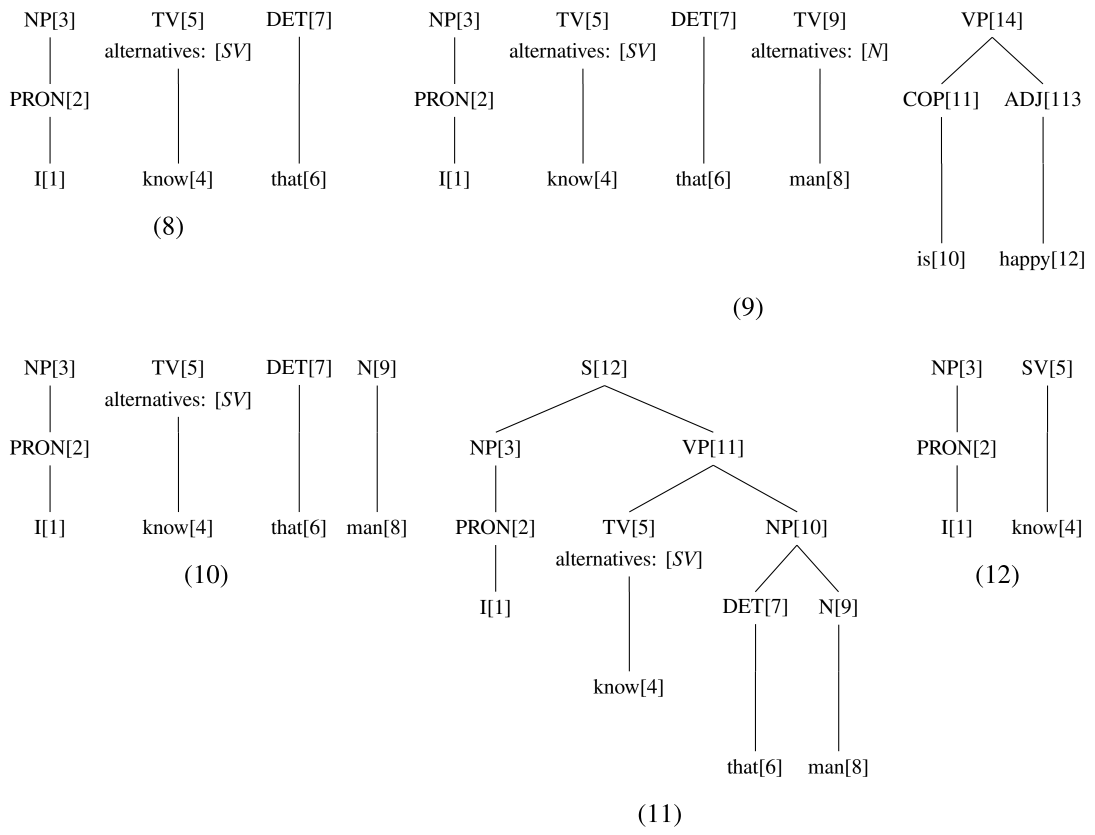


Figure 2.3: Bottom-up parse trees, from tree (8) to tree (12).

The algorithm has now produced a full parse tree but there are two more words remaining for processing (*is* and *happy*). Therefore, the parser backtracks to the last move where it has an alternative path to take, which is step 5 of Figure 2.3(11), where it re-analyses the input word *know* as *SV*, as shown in Figure 2.3(12).

Since there are no possible groupings of current analyses, the parser processes the subsequent input words and produces a number of analyses and partial parse trees (Detailed partial parse trees are included in Appendix A, from Figure A.2(18) to Figure A.2(22)) until it reaches a state as shown in Figure 2.3(13) where it cannot group any of the available partial parse trees and there are no more input words for the parser to explore. Then it backtracks to where it has an alternative path to explore, which is step 10 of Figure 2.3(13). This new analysis is shown in Figure 2.3(14).

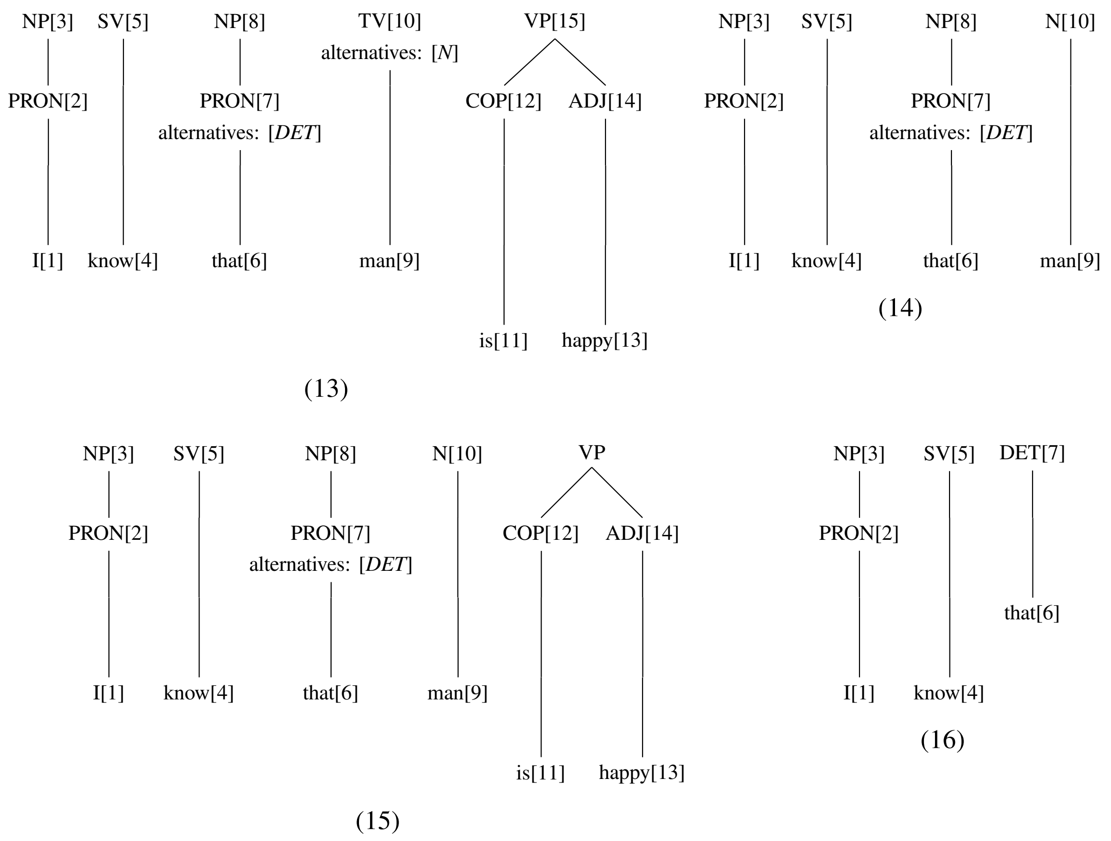


Figure 2.3: Bottom-up parse trees, from tree (13) to tree (16).

The algorithm then processes the input words *is* and *happy* and groups them together to make a new parse tree as in Figure 2.3(15). From the current partial parse trees it is not possible to group any of them. The parser backtracks to its last move where it has an alternative path to explore. The parser backtracks to step 7 of 2.3(15) by re-analysing the input word *that* as *DET*. This new partial parse tree is shown in 2.3(16).

Again, the algorithm performs several steps and produces a number of trees (These trees are shown in Appendix A, from Figure A.2(28) to Figure A.2(32))) before it reaches the state, as shown in the Figure 2.3(17), to group any of the partial parse trees and there are no more input words for processing. Hence, it backtracks to its last move where an alternative analysis is available, which is step 9 of Figure 2.3(17). This involves analysing the input word *man* as *N*, as shown in Figure 2.3(18).

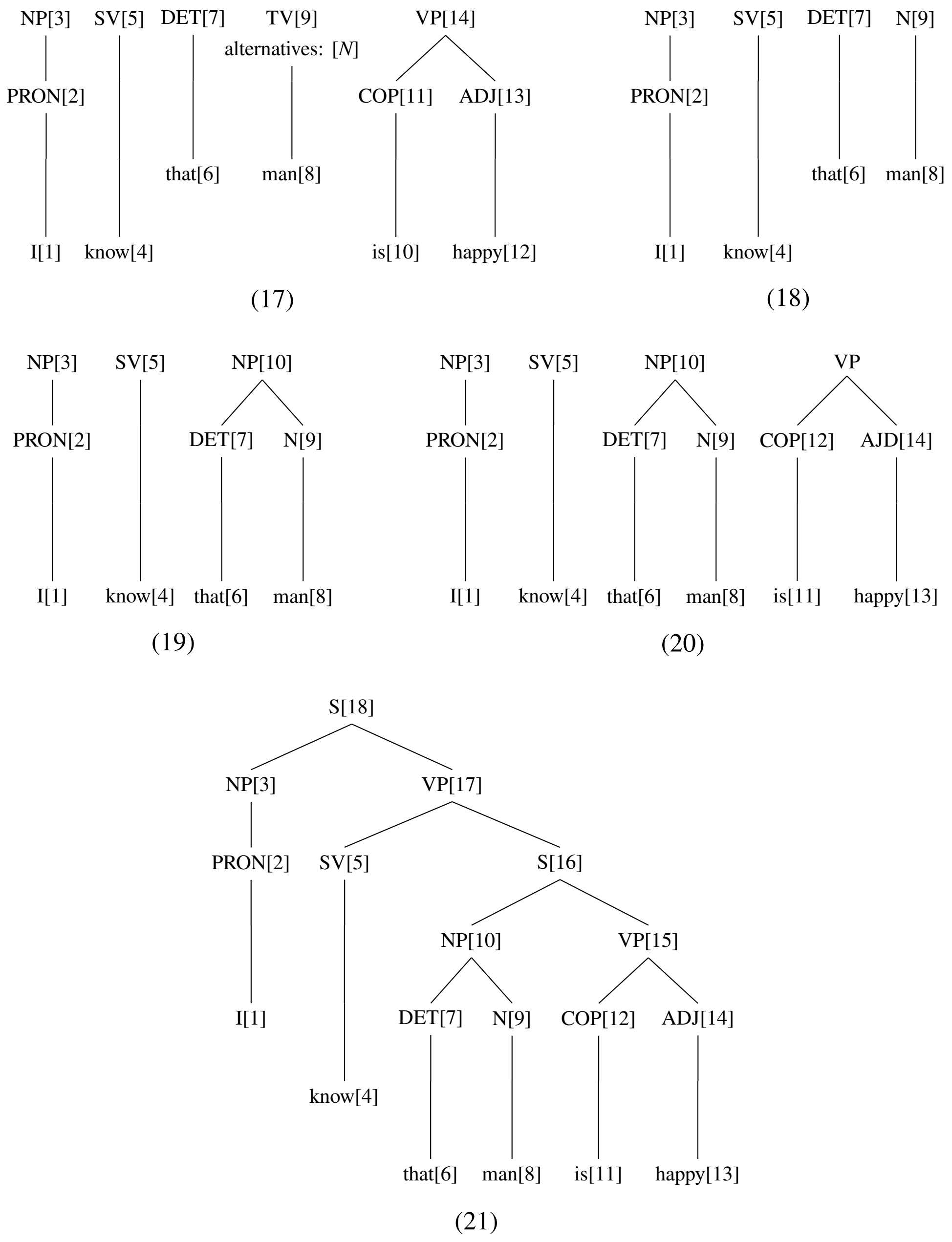


Figure 2.3: Bottom-up parser trees, from tree (17) to tree (21).

The algorithm then groups the *DET* and *N* of Figure 2.3(18) to produce an *NP* as in Figure 2.3(19). Then it analyses the input words *is* and *happy* for the production of *COP* and *ADJ* respectively, where it groups them for the production of *VP*, as in Figure 2.3(20).

The remaining partial trees are grouped under various non-terminal symbols (as in Appendix A, from Figure A.2(37) to Figure A.2(40)) until they are all grouped under the symbol *S* resulting in the production of the complete parse tree, as in Figure 2.3(21).

From the description of the parsing process above, backtracking can also occur in bottom-up parsing. The main cause of backtracking in bottom-up parsing is the presence of lexical ambiguities. For example, the word *man* from the sentence *I know that man is happy* has two lexical interpretations: (i) as a transitive verb (*TV*) and (ii) as a noun (*N*). Hence, it is possible that the parser may select the inappropriate interpretation for an input string, as it did when it produced an inappropriate interpretation of the word *man* (interpreted as *TV* instead of *N*) as shown in Figures 2.3(13) – which forced it to backtrack to select a more appropriate interpretation as shown in Figure 2.3(14).

A parsing algorithm that combines top-down and bottom-up parsing is left-corner parsing. In the next section, we will describe the way left-corner algorithm works. We will process the sentence *I know that man is happy* to understand the way this algorithm combines the two different algorithms, and to investigate whether it offers a solution to parser backtracking.

2.3 Left-corner parsing algorithm

In the previous two sections (Section 2.1 and 2.2) we have identified the main causes for backtracking in parsing which affect parsers' performance. In top-down parsing, backtracking occurs because parsers have no information about the lexical categories of input strings, where such information can guide parsers to make correct decisions when processing non-terminal symbols. In bottom-up parsing, backtracking occurs because parsers have no knowledge of non-terminal symbols, where such information can be used for avoiding arriving at dead ends when processing input strings.

Left-corner parsing combines top-down strategy with bottom-up strategy by mixing the steps of bottom-up parsing with those of top-down parsing. Such mixing helps a parser to make correct parsing decisions as the information becomes available.

The first element a left-corner parser investigates from a set of grammatical rules is the first symbol from the right-hand-side of a rule. For example, the *NP* is the left corner of the rule $S \rightarrow NP\, VP$ because it appears as the first symbol from the right-hand-side of the rule.

In order to understand the way the left-corner algorithm combines top-down and bottom-up algorithms, we will parse the sentence *I know that man is happy*, which we have used previously, by using this algorithm. Our aim in processing this sentence is to examine the way that this algorithm switches between steps of bottom-up parsing and top-down parsing and to find out if backtracking also occurs in this algorithm. We will use the same set of grammatical rules that we have used in the previous sections.

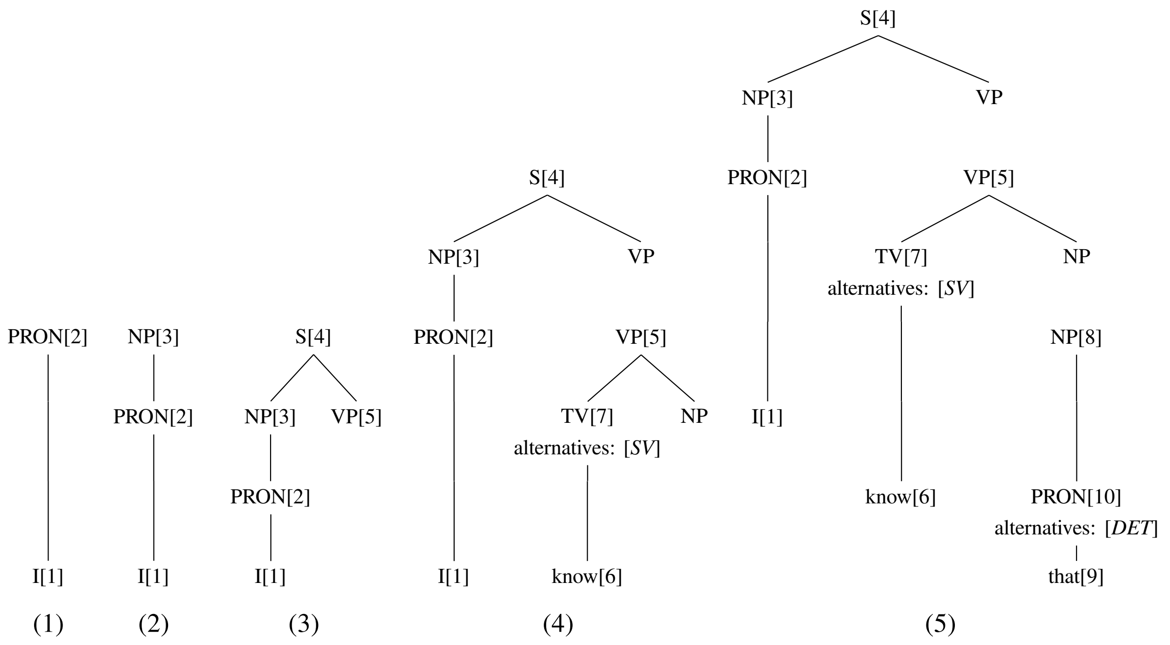


Figure 2.4: Left-corner parse trees, from tree (1) to tree (5).

The algorithm processes the first input string *I*, which is a bottom-up step, by using it for the production of *PRON*, as shown in Figure 2.4(1). Since *PRON* is the left-corner of the rule *NP* → *PRON*, the parser has now made a complete subtree of type *NP*, as shown in Figure 2.4(2). Moreover, the symbol *NP* matches the left corner of the rule $S \rightarrow NP\ VP$ so that a partial parse tree is produced, as shown in Figure 2.4(3).

The next move the parser takes is a bottom-up move consisting of analysing the next input string. The word *know* is analysed as *TV*, which is the first possible interpretation of it and is the left corner of the rule $VP \rightarrow TV\ NP$. A new partial parse tree is generated by using the symbol *TV* as the left-corner for the *VP*, as shown in Figure 2.4(4). Then, the parser takes a top-down strategy to process the *NP* symbol. The word (*that*) is parsed and it is checked whether it can be used for the production of *NP*. The symbol *PRON*, which is the first possible analysis for *that*, is the left corner of *NP* and is used for the production of the symbol *NP*, as shown in Figure 2.4(5).

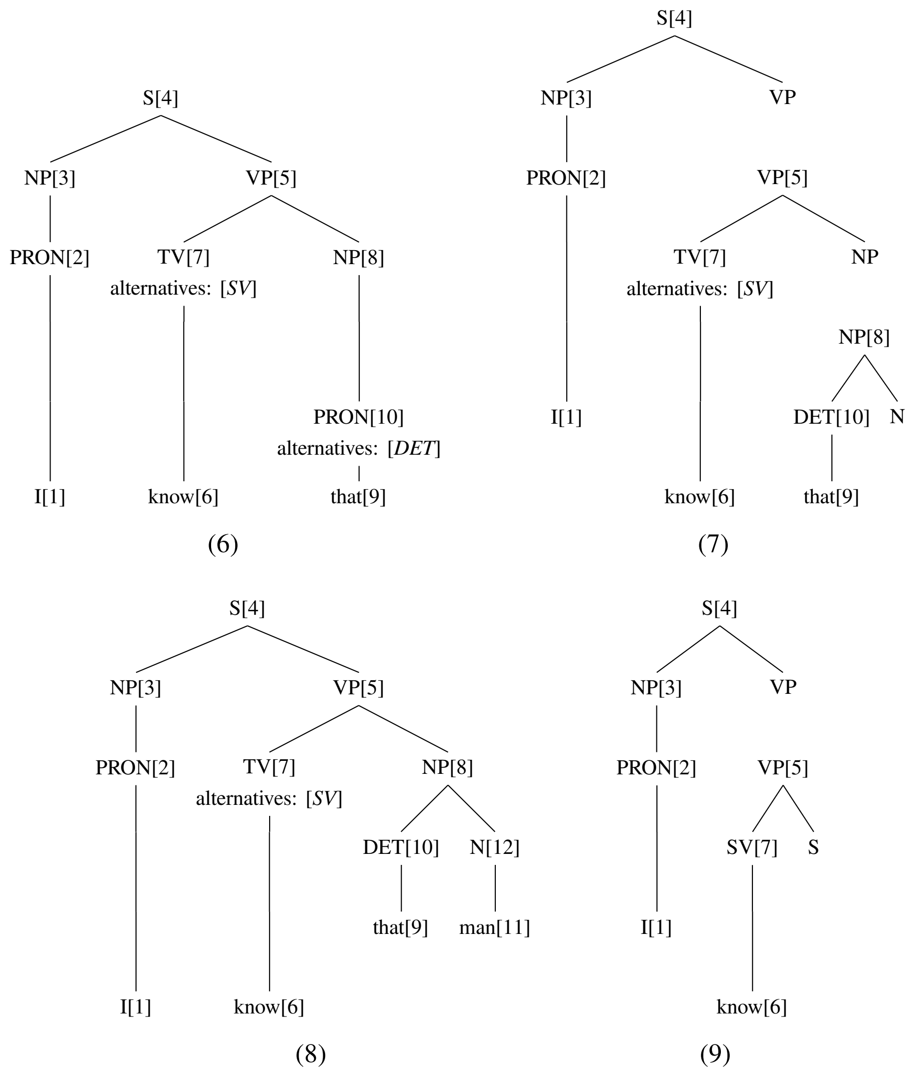


Figure 2.4: Left-corner parse trees, from tree (6) to tree (9).

Since the symbol *NP* is successfully produced and can be used for completing the subtree of *VP* where the subtree of *VP* completes the subtree of the symbol *S*, the parser can now produce a full parse tree, as shown in Figure 2.4(6).

However, there are three remaining input strings (*man*, *is*, and *happy*) to be processed. Thus, the algorithm backtracks to find an alternative analysis for recognising the symbol *VP*. So the parser backtracks to step 10 of Figure 2.4(6) and reprocesses the input string as *DET*, which is also the left corner of the *NP* and which generates a new partial parse tree, as shown in Figure 2.4(7).

The next input string *man* is processed as *N*, which completes the *NP* subtree, as shown in Figure 2.4(8), which is a complete parse tree. However, there are two remaining input strings (*is*, and *happy*) to be processed. Therefore, the parser backtracks to its most recent move where it had an alternative path to explore, which is step 7 of Figure 2.4(8) and reprocesses the input string *know* as *SV*, which is also the left corner of the symbol *VP* and which generates a new partial parse tree, as shown in Figure 2.4(9).

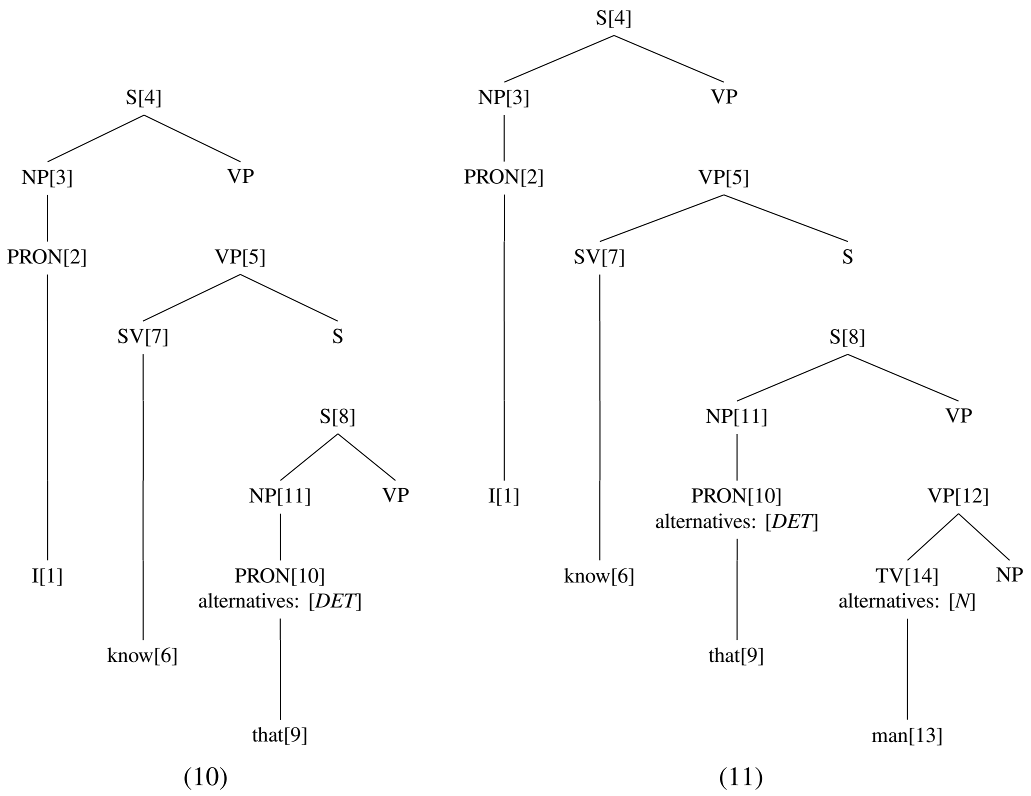


Figure 2.4: Left-corner parse trees from tree (10) to tree (11).

The next input string *that* is reprocessed as a *PRON* which is the left corner of the symbol *NP*, where *NP* is the left corner of the symbol *S*, so that a new partial parse tree is produced which satisfies the left corner of *S*. This tree is shown in Figure 2.4(10).

The next input string *man* is used for the production of the symbol *TV*, which is the left corner of the symbol *VP* and the new tree generated is shown in Figure 2.4(11). The remaining input strings (*is* and *happy*) cannot produce any productions that can be used as the left corner of the *NP* symbol. Hence, the parser backtracks to its last move to use an alternative choice, which is interpreting man as *N* even though *N* is not the left corner of *VP*. The parser backtracks further to step 10 and reinterprets *that* as *DET*. The symbol *DET* is the left corner of the symbol *NP* and a new partial parse tree is generated as in Figure 2.4(12). A possible interpretation of the input string *man* is *N*, which can complete the subtree *NP* as in Figure 2.4(13). The newly produced *NP* is then used as the left corner of the symbol *S*, as shown in Figure 2.4(14).

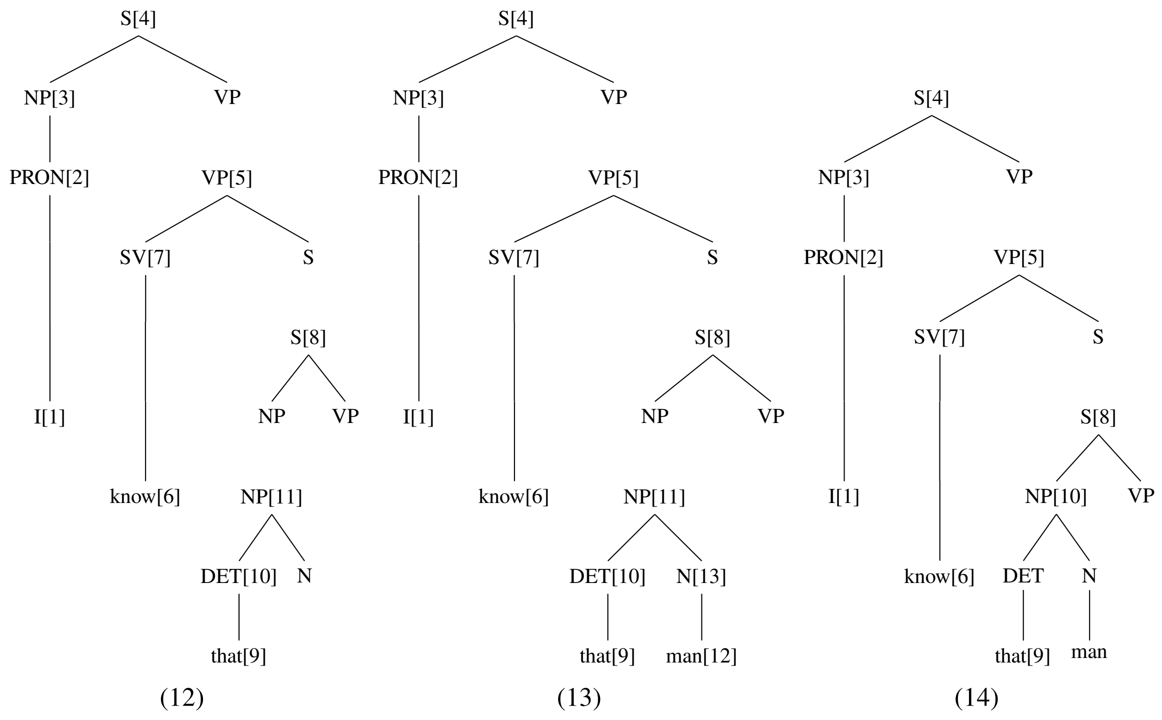


Figure 2.4: Left-corner parse trees, from tree (12) to tree (14).

The symbol *VP* is then processed. The input string is analysed as *COP*, which is the left corner of the symbol *VP*, as shown in Figure 2.4(15). Next, the input string *happy* is used for the production of the symbol *ADJ*, as shown in Figure 2.4(16).

The algorithm then plugs in the current subtrees. This newly generated tree is then used to complete the subtree of the *VP* which is shown in Figure A.3(17) in Appendix A. The complete subtree for the symbol *VP* is used to complete the subtree *S*, as shown in Figure A.3(18) in Appendix A, where the complete subtree of *S* is used for completing the subtree *VP* as in Figure A.3(19) in Appendix A. Finally, the full parse tree is then completed by plugging in the complete subtree of *VP*, as shown in Figure 2.4(17).

Although left-corner parsing combines top-down and bottom-up parsing strategies, however, it cannot overcome the problem of backtracking.

People have used various techniques to overcome parser backtracking. Kay [Kay, 1973] followed by Ramsay [Ramsay, 1980] and Norvig [Norvig, 1991] has solved the problem of exponential time complexity in top-down parsers. They have constructed sets of mutually recursive functions using chart tables. The basic idea of such approaches is that when a parser is applied to the input, the result is stored in a chart

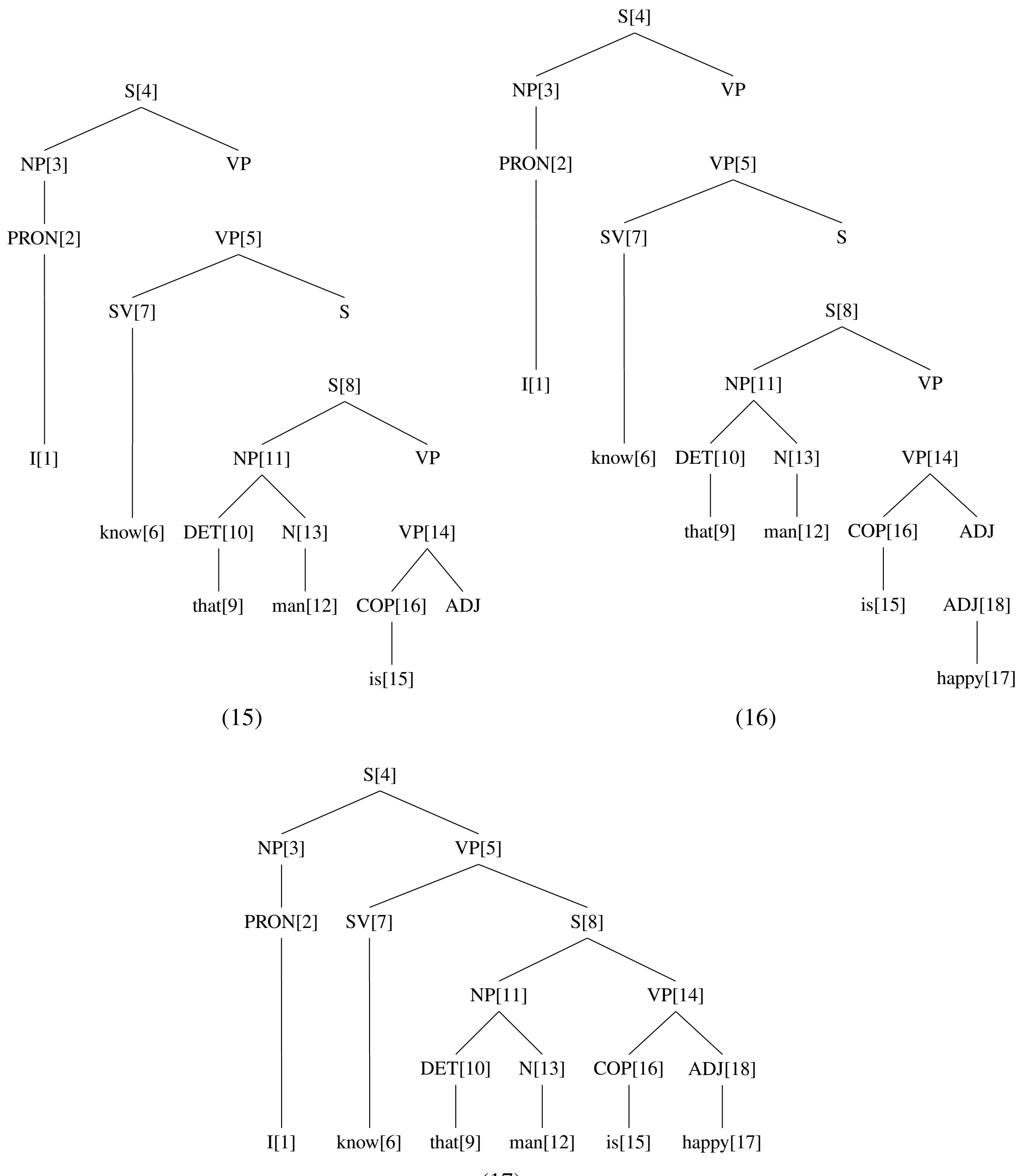


Figure 2.4: Left-corner parse trees, from tree (15) to tree (17).

table for subsequent reuse if the same parser is ever reapplied to the same input. These notions have been unified into an approach known as chart parsing, which is described in the next section.

2.4 Chart parsing algorithm

The notion of tabular/chart parsing has been introduced independently by different people, e.g. [Kasami, 1965, Younger, 1967, Cocke and Schwartz, 1970, Earley, 1970, Kay, 1973]. Chart parsing is often used as a general framework for constructing parsing algorithms [Pereira and Warren, 1983]. It is described in the literature as a technique for constructing a full representation of all the possible analyses for an input string [Kaplan, 1973]. The adaptation of chart parsing from constituent-based algorithms, which is a modification of phrase-structure chart parsing, to dependency parsing has been described by Meixun et al., [Meixun et al., 2011].

A chart can be thought of as a network wherein nodes represent the spaces between different elements of an input string, and labelled arcs represent the lexical categories of the words which span from one node to another. Chart parsers allow simple representations of alternative analyses. Hence, a chart parser produces a number of partial analyses and uses them in subsequent parsing steps [Yamada and Matsumoto, 2003].

We will demonstrate the theory of chart parsing by analysing the sentence *I know that man is happy* using the grammar rules in Figure 2.5. The chart parser introduces nodes for each word in the given sentence. The nodes are connected by arcs and the arcs are labelled with the lexical categories for the words. Analyses produced by the parser are stored in a chart, where the parser may reuse them at later stages during parsing.

```
S       →    NP   VP
NP      →    PRON
NP      →    DET  N
VP      →    COP  ADJ
VP      →    TV   NP
VP      →    SV   S
PRON    →    I
TV      →    know
SV      →    know
PRON    →    that
DET     →    that
TV      →    man
N       →    man
COP     →    is
ADJ     →    happy
```

Figure 2.5: Context-free grammar rules for parsing.

In this section, we will process the sentence *I know that man is happy* in order to illustrate the operation of chart parsers and identify the way this algorithm avoids backtracking.

The parser starts by analysing the first word *I*. It produces an arc spanning from node 1 to node 2, as shown in Figure 2.6(1) (The arcs coloured in red are the most recently created ones). Then, where *PRON* can be used as the production of the symbol *NP*, a new arc is generated with the *NP* as its label, as shown in Figure 2.6(2). Since there are no more analyses for the input string *I* and the parser cannot produce new arcs, it will move to process the next input string (*know*), where a new arc is generated, spanning the distance from the second node and the third node in the chart graph, as shown in Figure 2.6(3). Since there is a different lexical interpretation for the word *know*, which is *SV*, the parser creates a new arc from node 3 to node 4 for the word with the *SV* label, as shown in Figure 2.6(4)[7].

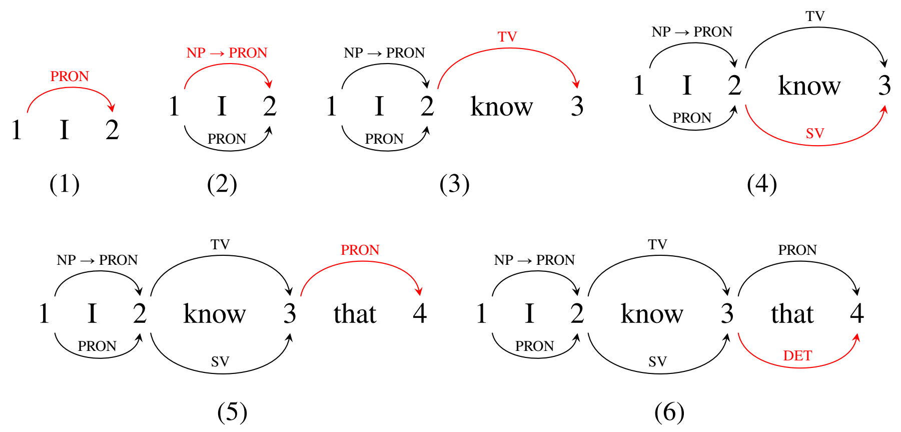
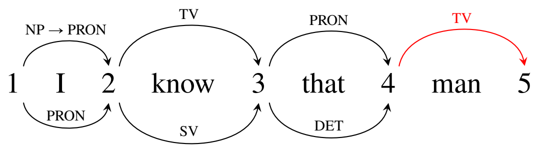


Figure 2.6: Chart parsing graphs, from graph (1) to graph (6).

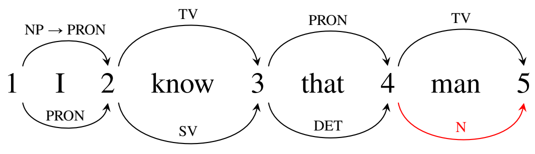
Moreover, the word *that* is analysed and given two interpretations since it can be *PRON* or *DET*, according to the grammatical rules in Figure 2.5. The newly created arcs from node 3 to 4 are then added to the chart, as in 2.6(5) and 2.6(6).

[7]The presentation of the arcs either below or above the words does not have any significance. They are presented in this way for the clarity of the graphs only.

Furthermore, the word *man* is analysed and given two interpretations since it can be *TV* or *N*. The newly created arcs from node 4 to 5 are added to the chart, as in 2.6(7) and 2.6(8).



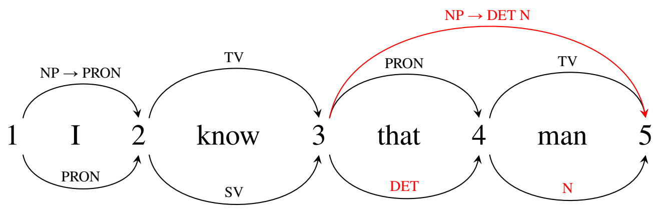
(7)



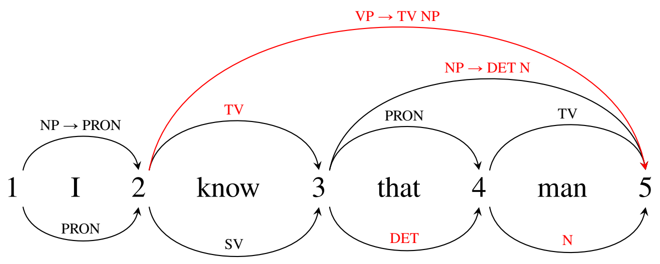
(8)

Figure 2.6: Chart parsing graphs (7) and (8).

Newly labelled arcs are built over the nodes progressively and the process of adding new arcs to the chart is repeated until there are no more labelled arcs to be added. The parser attempts to create new arcs from the arcs that already exist in the chart, so that the parser can create a new arc spanning from node 3 to node 5 and label it with *NP*, because the combination of symbols *DET* (from node 3 to node 4) and *N* (from node 4 to node 5) can be achieved. The new arc is then added to the chart, as in Figure 2.6(9). Additionally, the parser generates a new arc spanning from node 2 to node 5 with a label *VP* because combining a *TV* with an *NP* can produce a *VP*. This new chart is shown in Figure 2.6(10).



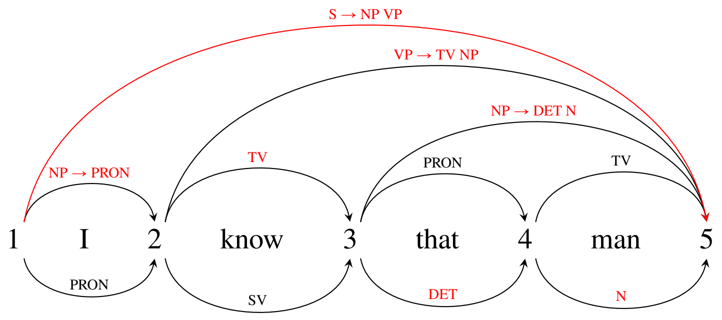
(9)



(10)

Figure 2.6: Chart parsing graphs (9) and (10).

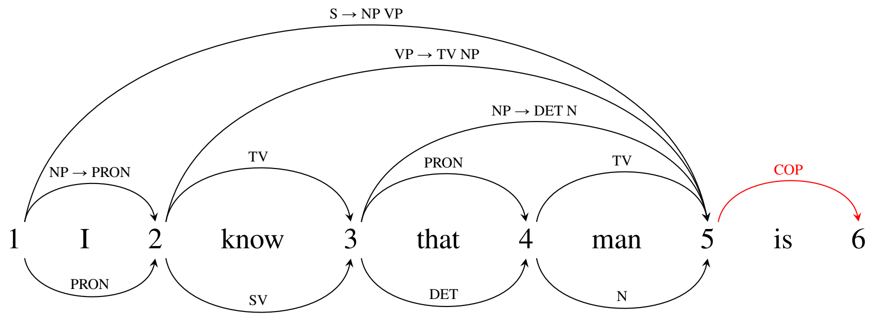
Finally, the algorithm creates a new arc labelled *S* spanning from node 1 to node 5 as in Figure 2.6(11).



(11)

Figure 2.6: Chart parsing graph (11).

The algorithm can then declare the sentence *I know that man* as a well-formed sentence because according to the grammatical rules in Figure 2.5, all the analyses of the words lead to the production of a labelled arc with S. However, the arc with the label *S* does not span over the entire length of the sentence because there are still more words to be processed. The parser then processes the next available input string, which is *is* and generates a new arc for it with label *COP*, as shown in Figure 2.6(12), where it spans from node 5 to node 6.

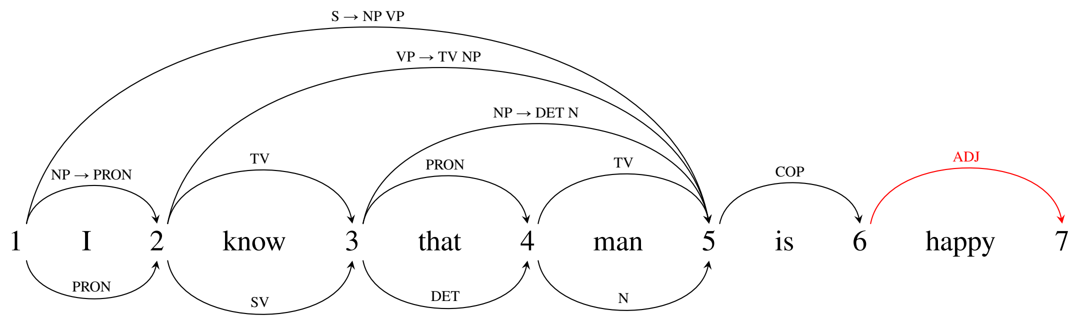


(12)

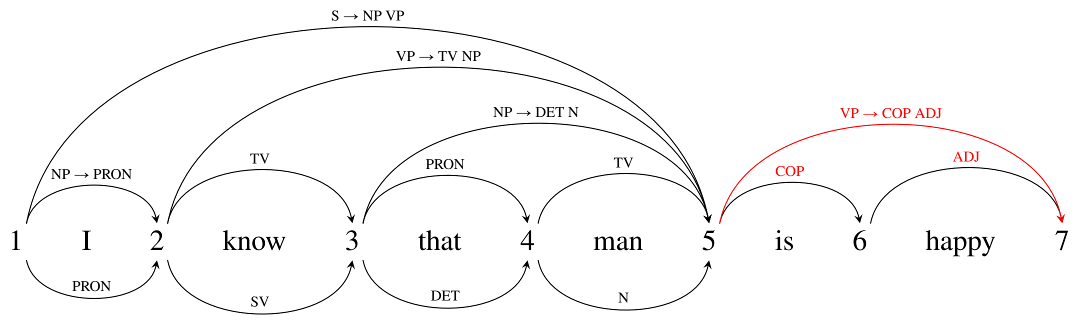
Figure 2.6: Chart parsing graph (12).

Next, the final word *happy* is processed and a new arc is generated for it, which spans the distance from node 6 to node 7, as shown in Figure 2.6(13).

From the newly created arcs it is also possible to produce a new arc with the symbol *VP*, where the symbol *VP* is made by using the symbols *COP* and *ADJ*, by spanning it from node 5 to node 7, as shown in Figure 2.6(14).



(13)



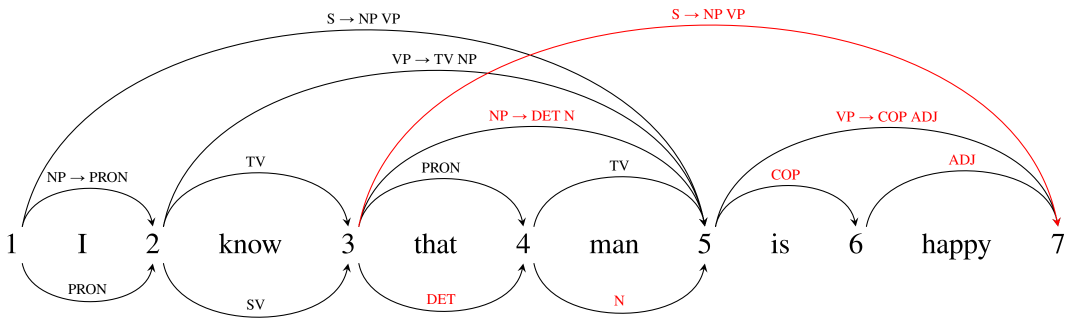
(14)

Figure 2.6: Chart parsing graphs, from graph (13) to graph (14).

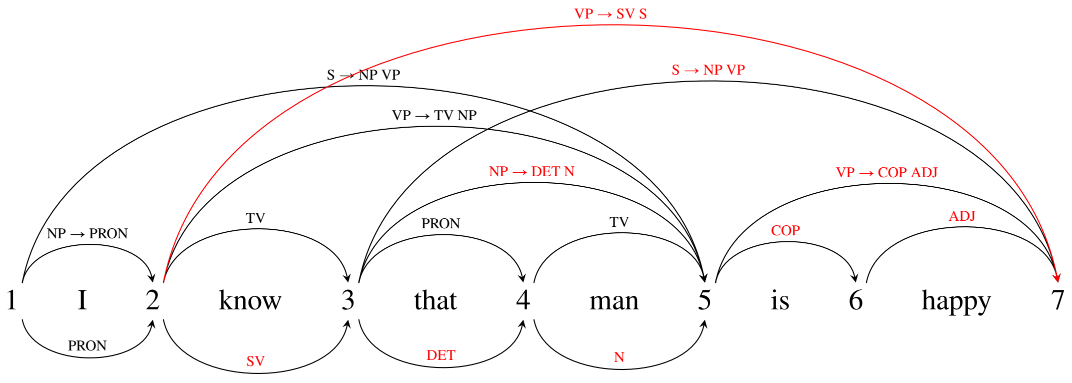
Furthermore, from the available arcs (the arc spanning from node 3 to 5 with label *NP* and the arc spanning from node 5 to 7 with label *VP*) in the chart, the parser generates a new arc with the label *S* spanning from node 3 to node 7, as shown in Figure 2.6(15). Next, using the arc with the label *SV*, which spans the distance from node 2 to node 3, and using the newly created arc with label *S* that spans node 3 to node 7, a new arc is generated with the label *VP* spanning from node 2 to node 7, as in Figure 2.6(16).

Finally, by using the arc labelled *NP* that was generated initially for the word *I*, and the arc labelled *VP* spanning from node 2 to node 7, a new and final arc is created with the label *S*, which covers the entire length of the sentence, as shown in Figure 2.6(17).

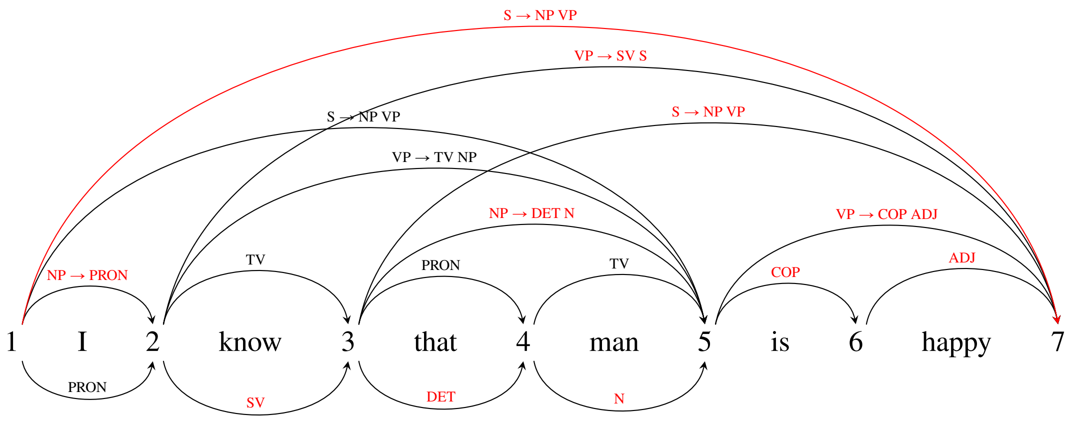
A number of the partial analyses that are produced may not contribute to the final parsing solution. However, they will be available throughout the parsing process and can be used later if they are required for the final parsing solution. The advantage of using chart parsing is that parsers do not rediscover paths that have already been explored.



(15)



(16)



(17)

Figure 2.6: Chart parsing graphs, from graph (15) to graph (17).

2.5 Shift-reduce parsing algorithm

A shift-reduce algorithm consists of two data structures: (i) a queue which consists of items that have not yet been inspected, and (ii) a stack which consists of items that have been looked at but have not been used as part of a larger structure. Shift-reduce parsers can perform one of two actions at any time, which are known as shifting or reducing.

SHIFT: This action moves the head of the queue onto the top of the stack. This action is taken in one of two situations: (i) if the head of the queue does not match the right-hand-side of a rule and the stack is empty, and (ii) if the head of the queue and the items on the top of the stack do not match the right-hand-side of a rule.

REDUCE: If the head of the queue, or the head of the queue and the item on the top of the stack match the right-hand-side of a rule then the head of the queue or the head of the queue and the item on the top of the stack are removed and the left-hand-side element of the rule is placed at the head of the queue.

The parsing process is successful if the queue is empty and the stack contains one symbol which satisfies the sentential condition [8]. However, the parsing process fails if the queue is empty and the last item on the stack does not satisfy the sentential condition and the parser has exhausted all possible analyses for all of the input strings.

In this section, we will use the grammatical rules shown in Figure 2.7 to parse the sentence *I know that man is happy* by using the shift-reduce parsing strategy. Some of the steps that the parser takes are included in Figure 2.8. The full parsing steps are included in Appendix B.

[8]Sentential conditions are considered satisfactory if they conform to a given grammar rule that defines the structure of a sentence, such as $S \rightarrow NP\ VP$, which means that a valid sentence may consist of an *NP* followed by a *VP*.

S	→	NP VP
NP	→	PRON
NP	→	DET N
VP	→	COP ADJ
VP	→	TV NP
VP	→	SV S
PRON	→	I
TV	→	know
SV	→	know
PRON	→	that
DET	→	that
TV	→	man
N	→	man
COP	→	is
ADJ	→	happy

Figure 2.7: Context-free grammar rules for parsing.

The algorithm first places the sentence *I know that man is happy* in a queue and creates a stack of empty categories as in step 1 in Figure 2.8.

After this, the head of the queue, which is *I*, matches the right-hand-side of the rule *PRON → I* so it is replaced with the symbol *PRON*, as shown in step 2 in Figure 2.8. The symbol *PRON* at the head of the queue which matches the right-hand-side of the rule *NP → PRON* is replaced with the symbol *NP* and the new symbol is placed at the head of the queue. Since the head of the queue does not match the right-hand-side of any rules and the stack is empty, the only action the parser can take is to do a SHIFT. This results in shrinking the queue by placing the head of the queue, which is the symbol *NP*, on the top of the stack as shown in step 4 of Figure 2.8.

From step 4, one of the interpretations for the input string *know* is *TV* so it is reduced to *TV* while the second interpretation of it, which is *SV*, is used to create a backtracking point in order to explore it at later stages if necessary. After step 6, a number of SHIFT and REDUCE actions are performed and new backtracking points are created along the way, as shown in steps 5 to 18, until the parser is in a situation where it has an empty queue and a non-empty stack and a sentential condition is not being met, as in step 18. In step 18, the parser backtracks to its last move where it had an alternative path to explore, which is step 11 where it can re-analyse the input string *man* as *N*.

The subsequent steps from step 19 involve a number of SHIFT and reduce actions.

```
Steps Queue                          Stack                Action                    Alternatives
-------------------------------------------------------------------------------------------------
1     I know that man is happy       []                   REDUCE: PRON → I          -
2     PRON know that man is happy    []                   REDUCE: NP → PRON         -
3     NP know that man is happy      []                   SHIFT                     -
4     know that man is happy         NP                   REDUCE: TV → know         SV → know
5     TV that man is happy           NP                   SHIFT                     -
6     that man is happy              TV NP                REDUCE: PRON → that       DET → that
...
10    S man is happy                 []                   SHIFT                     -
11    man is happy                   S                    REDUCE: TV → man          N → man
...
18    []                             VP TV S              Backtrack to step 11      -
19    man is happy                   S                    REDUCE: N → man           -
20    N is happy                     S                    SHIFT                     -
...
26    []                             VP N S               Backtrack to step 6       -
27    that man is happy              TV NP                REDUCE: DET → that        -
28    DET man is happy               TV NP                SHIFT                     -
29    man is happy                   DET TV NP            REDUCE: TV → man          N → man
...
36    []                             VP TV DET TV NP      Backtrack to step 29      -
37    man is happy                   DET TV NP            REDUCE: N → man           -
...
47    []                             VP S                 Backtrack to step 4       -
48    know that man is happy         NP                   REDUCE: SV → know         -
49    SV that man is happy           NP                   SHIFT                     -
50    that man is happy              SV NP                REDUCE: PRON → that       DET → that
51    PRON man is happy              SV NP                REDUCE: NP → PRON         -
52    NP man is happy                SV NP                SHIFT                     -
53    man is happy                   NP SV NP             REDUCE: TV → man          N - man
...
60    []                             VP TV NP SV NP       Backtrack to step 53      -
61    man is happy                   NP SV NP             REDUCE N → man            -
62    N is happy                     NP SV NP             SHIFT                     -
...
68    []                             VP N NP SV NP        Backtrack to step 50      -
69    that man is happy              SV NP                REDUCE: DET → that        -
70    DET man is happy               SV NP                SHIFT                     -
71    man is happy                   DET SV NP            REDUCE: TV - man          N → man
72    TV is happy                    DET SV NP            SHIFT                     -
...
78    []                             VP TV DET TV NP      Backtrack to step 71      -
79    man is happy                   DET SV NP            REDUCE: N → man           -
80    N is happy                     DET SV NP            REDUCE: NP → DET N        -
...
86    VP                             NP SV NP             REDUCE: S → NP VP         -
87    S                              SV NP                REDUCE: VP → SV S         -
88    VP                             NP                   REDUCE: S → NP VP         -
89    S                              []                   SHIFT                     -
90    []                             S                    -                         -
```

Figure 2.8: Shift-reduce parsing steps.

But, the parser has to backtrack to step 6 because in step 26 it is in a situation where it has an empty queue and a stack of items that do not satisfy the sentential condition. In step 27, the input string *that* is re-analysed as *DET* and then is shifted onto the top of the stack because no reduce action is possible. The word *man* is then re-analysed as *TV* and its alternative analysis, which is *N*, is marked as a backtracking point, as shown in step 29 where the parser backtracks to it after several parsing processes, as shown in step 36. This recent backtracking also leads to a dead end path at step 47, where the parser backtracks further to step 4 because this was its last move where it can reinterpret the input string *know* as *SV*. Parsing from step 48 continues, but this time the alternative analysis (*SV*) is used for the head of the queue (*know*). Several SHIFT and REDUCE actions are performed between steps 48 and 60 and new backtracking points, such as those in step 50 and 53, are created. In step 60, the parser is forced to backtrack to step 53 to re-analyse the input string *man* as *N* and then it continues until it gets to step 68, where it backtracks again to step 50 to re-analyses the input string *that* as *DET* as shown in step 70.

From step 71, the input string *man* is interpreted as *TV* and a backtracking point is created for its alternative interpretation if it fails again. At step 78, the parser fails and it then explores the alternative interpretation for the input string *man*, which is *N*, as shown in step 79. From step 80, the head of the queue and the top of the stack matches the right-hand-side of the rule $NP \rightarrow DET\ N$, where they are reduced and the symbol *NP* is placed at the head of the queue and the symbols *DET* and *N* are removed from the stack and the queue. Between steps 80 and 86 a number of SHIFT and REDUCE actions are performed. In step 86, the head of the queue (*VP*) and the top of the stack (*NP*) matches the right-had-side of the rule $S \rightarrow NP\ VP$. They are then reduced to *S* and the symbol *S* is placed at the head of the queue, as shown in step 87. At this stage, the parser has an *S* as the head of the queue and an *SV* on the top of the stack, which can create a symbol *VP* by combining the *SV* with the *S*, which then places the new symbol (*VP*) in the queue as in step 88.

The last remaining symbols *VP* from the queue and *NP* on the stack are then combined to produce the symbol *S*, where the symbol *S* is then shifted on to the stack, as shown in steps 89 and 90 respectively. Finally, there are no items in the queue, and only one item on the stack which is *S*, which indicates that the sentence *I know that man is happy* is a valid sentence according to the grammatical rules in Figure 2.7.

Shift-reduce algorithm also use backtracking as long as alternative paths are available to choose from at any point during the parsing process. Backtracking occurs in shift-reduce parsing for the same reason as in the other parsing algorithms that we have described above, namely the presence of different lexical interpretation of words which create ambiguities in sentences. However, shift-reduce parsing algorithms play an important role in dependency parsers such as MaltParser [Kuhlmann and Nivre, 2010], which is similar to the type of the parser we are trying to develop in this project.

The above procedure of the shift-reduce parsing is one way of using it. Other parsing systems may perform SHIFT and REDUCE in different ways, for example, where dealing with long distance dependency between words in natural language sentences, one might attempt to combine the head of the queue with an item buried inside the stack to maximise the possibility of performing a reduce operation.

2.6 Summary

In this chapter we have described a number of different algorithms. These algorithms can be applied in various ways to parsing natural languages.

We have identified that one of the major issues with parsing, which may affect parsing speed at least, is the backtracking of parsers, which is often caused by local ambiguities in natural language sentences. As we have already mentioned, during backtracking, parsers lose all previously successful analyses of input strings and they will re-analyse them. This affects parsing performance in terms of speed, and potentially failing parsers to produce correct results after all. However, we have also identified that one way of overcoming the problem of backtracking is by using chart parsing, as described in Section 2.4, where the parser produces multiple analyses for ambiguous input strings and then attempts to produce new analyses from the partial analyses in order to generate one final analysis that can represent the complete sentence.

From the above investigation of parsing algorithms, we would like to be able to merge features of chart parsing with shift-reduce parsing[9]so that we can benefit from them by producing a fast, robust and accurate parser. We base our implementation of our parser on the arc-standard algorithm[10] of MaltParser [Kuhlmann and Nivre,

[9]Since shift-reduce parsing deals with lexical items it is effectively a bottom-up process.

[10]This is a variation of MaltParser, which is based on shift-reduce algorithm. SHIFT corresponds to the SHIFT action in shift-reduce algorithm. LEFT-ARC and RIGHT-ARC correspond to the REDUCE action. LEFT-ARC makes the head of the queue the parent of the topmost item on the stack. RIGHT-ARC makes the topmost item on the stack the parent of the head of the queue.

2010] which is based on a shift-reduce algorithm. Shift-reduce algorithms have contributed to the development of a number of fast, accurate, and robust parsers [Zhu et al., 2013, Kuhlmann and Nivre, 2010, Attardi, 2006, Ratnaparkhi, 1999]. We apply some extensions to the arc-standard algorithm by using features of chart parsing, where we store parse analyses in a chart table in order to reuse them when necessary.

In this chapter, we have focused on investigating the general application of different parsing algorithms to natural language texts. Historically, these algorithms have used grammatical rules for determining the relationships between lexical categories of a set of input strings. As we have shown in the examples, these kinds of approaches can be referred to as grammar-driven approaches.

However, in recent years, there has been an increasing interest in using machine learning algorithms for parser training and identifying relationships between lexical categories of a set of input string. Parsers using this kind of approach are called data-driven approaches.

Furthermore, both dependency frameworks and phrase structure frameworks can be used in this context. In the following chapter, we will investigate how these two widely different frameworks can be used with any of the above algorithms for the parsing of natural language texts.

Chapter 3

Natural Language Parsing Frameworks

There are two widely used different frameworks that we can use for the application of the parsing algorithms that we discussed in Chapter 2 for processing natural language sentences. These are: (i) a dependency framework (DF), and (ii) a phrase structure framework (PSF). In a DF, syntactic relations between words in sentences are based on the notion of dependencies, where a word depends on a different word that may or may not be adjacent to it. In PSF, the notion of constituency is used for expressing relationships between words in sentences, where one or more lexical category of words is grouped under one category to form a phrase.

Melčuk [Melčuk, 1988, Melčuk, 1979], points out that DF differs from PSF in the way that non-terminal symbols, which are heavily used in phrase structure analyses, are not present in DF because a dependency structure is built from binary relations between words.

The use of syntactic categorisation is an integral part of syntactic representation in PSF but relationships among units (words) are not stated explicitly. In a DF representation, the types of syntactic relationships can easily be specified, but syntactic categorisation of units is not stated directly within the syntactic representation itself, and thus dependency frameworks do not use non-terminal symbols in dependency representations. Instead, items are related to each other by syntactic relationships (e.g. subjective, determinative, prepositional, etc.) The sentence *the cat sat on the mat* can have the phrase structure as shown in Figure 3.1(1) and the dependency structure as shown in Figure 3.1(2).

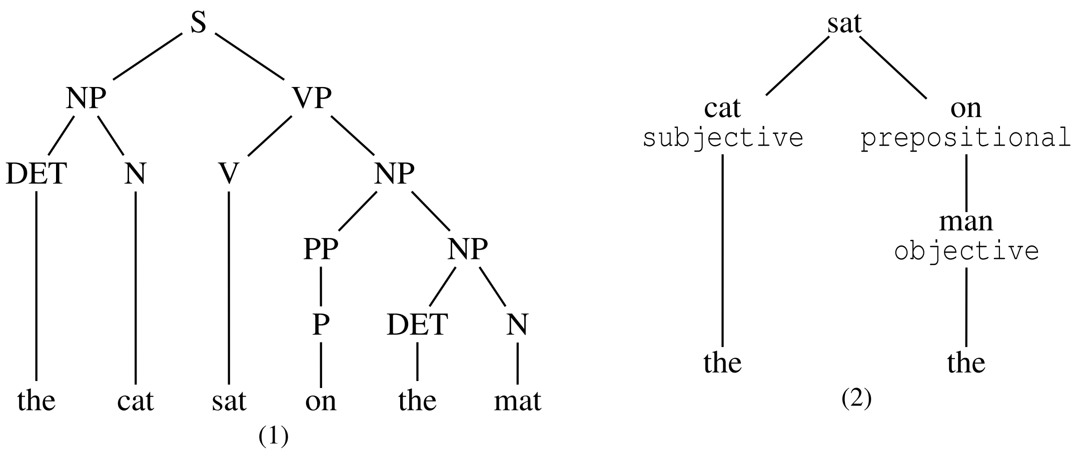


Figure 3.1: Phrase structure and dependency trees for the sentence *the cat sat on the mat*.

Despite the differences between these two frameworks in terms of syntactic representation they share some similarities. For example, both frameworks can use the notion of head. In dependency frameworks, a word is the head of another word. And in many phrase structure frameworks, each phrase has a head that determines the main properties of the phrase.

In this chapter, we will briefly highlight some of the main features of these two frameworks. We will also provide an overview of some of the state-of-the-art parsers that are based on these frameworks.

Our goal in this chapter is to study these two different frameworks and identify their features so that we can use them for constraining a data-driven parser to follow the parsing routes that are more likely to produce a correct analysis for a given sentence.

3.1 Dependency framework

Recent trends in using dependency framework for parsing natural language syntax have led to the development of a number of state-of-the-art parsers. It has been noted by McDonald et al., [McDonald et al., 2006], Nivre [Nivre, 2005] and Debusmann [Debusmann, 2000] that the starting point of modern linguistic theories in dependency framework goes back to the work of Tesnière [Tesnière, 1959].

Generally, in dependency based formalisms, the syntactic structure of a sentence consists of binary asymmetrical relations between the words in a sentence [Tesnière, 1959]. The notion is that in a sentence, all words depend on other words except one word, which is called the root word; i.e., the root word does not depend on any other words (sometimes people use an artificial word called root and assign it as the head of the main word in a sentence for convenience). The syntactic structure with binary asymmetrical relations for the sentence *the cat sat on the mat* is demonstrated in Figure 3.2.

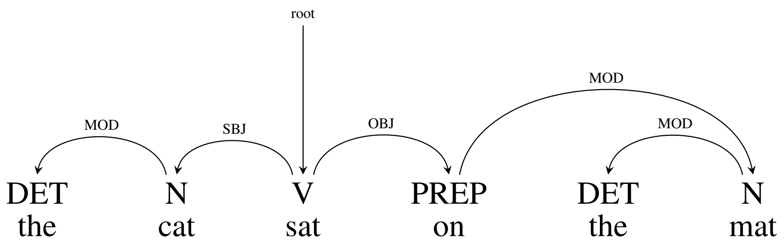


Figure 3.2: Dependency graph for the sentence *the cat sat on the mat*.

Another aspect of dependency graphs is that dependency relations between words are based on grammatical functions, for example, a word depends on another word either if it is a complement or a modifier of the latter. For instance, a transitive verb, such as love, requires two arguments where one argument has a grammatical function of being the subject and the second one has a grammatical function object, as in 4, *John*, which as a *noun* has the grammatical function of being a subject, while *Mary* has the grammatical function of being an object.

(4) John loves Mary

A number of studies in the theoretical linguistics community on dependency representations have followed on from the work of Tesnière. These studies resulted in a large and diverse set of grammatical theories and formalisms of dependency representations. These include Functional Generative Description [Sgall et al., 1986] and Dependency Unification Grammar [Hellwig, 2003, Hellwig, 1986]. Additionally, some of the other variations of dependency grammar include Constraint Dependency Grammar [Harper and Helzerman, 1995], Weighted Constraint Dependency Grammar [Schröder, 2002], and Functional Dependency Grammar [Tapanainen and Järvinen, 1997]. Furthermore, in Dependency Grammar Logic [Kruijff, 2001] a combination of dependency grammar and categorial grammar can be found. Nivre [Nivre, 2006] provides a detailed discussion of the history of dependency grammar, on which we have based this section.

Popular theories of dependency grammar include Melčuk's Meaning Text Theory (MTT) [Melčuk, 1988] and Hudson's Word Grammar (WG) [Hudson, 1984]. It is not possible to cover all of these theories in any detail here. However, we will attempt to draw on some of the theoretical background of this framework and explicitly mention certain dependency based formalisms when possible.

3.1.1 Theoretical background

3.1.1.1 Acyclic and labelling

Since the syntactic relations between words in sentences are based on binary relations, some restrictions are imposed on them as stated by Melčuk [Melčuk, 1979]:

1. Antisymmetric: The syntactic relationships between items are directed (i.e., two items cannot be the dependents of each other), e.g., if X depends on Y, then Y cannot depend on X. Formally, if X → Y then $\neg Y \rightarrow$ X.

2. Acyclic: An item cannot relate to itself, i.e., an item cannot depend on itself, it must depend on a different item. Formally, $\neg X \rightarrow X$ or $\neg X \circlearrowleft$.

3. Labelled relations: Syntactic dependency relations between items must be distinguished by using labels, as we have shown in Figure 3.2.

4. Head: Each word has at most one head.

5. Root: One word (the main word in the sentence) has no head.

3.1.1.2 The notion of head

Whenever a dependency relation is established between words, one word becomes the head of another word and the word receiving the head becomes the dependent word. It is possible, and it is allowed in dependency frameworks, that a word may become the head of a number of other words in a sentences, as we have shown in Figure 3.2 where the word *sat* is the head of *cat* and is also the head of *on*.

However, as noted above in most variations of dependency framework, a word cannot have more than one head; i.e., a word may have multiple dependents but a dependent cannot have multiple heads. However, in word grammar (WG), as developed by Hudson [Hudson, 1984], the concept of modifier sharing is introduced, where a dependent may be eligible for multiple heads or modifiers. Hudson [Hudson, 1984, p. 83] argues that the subject (*Mary*) of verb *seem* in Figure 3.3 can also be used (syntactically) as the subject of the infinitive although semantically it could be argued that the subject (*Mary*) is related only to the infinitive and not at all to *seem*[1]. There are other situations introducing multiple heads such as embedded interrogative clauses. For more discussion and examples of these situations, see Hudson [Hudson, 1984, p. 82–85].

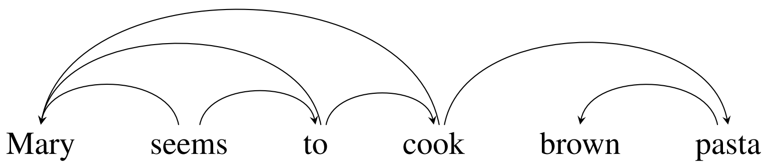


Figure 3.3: WG dependency structure.

3.1.1.3 The notion of single root item

In a dependency structure, every word should have a head. This means that the main word in a sentence must also have a head on its own. But, in any given sentence, there may not be a word to be the head of the main word. For example in the sentence *John ate pasta*, the word *ate* is the main word in the sentence because it is the head of *John* and *pasta*. And the dependents (*John* and *pasta*) cannot act as the head of *ate* because a dependent cannot be the head of its own head (antisymmetric). Since no words in the sentence can be the head of the main word, a dummy word (*root*) is assigned to the main word, as shown in Figure 3.4.

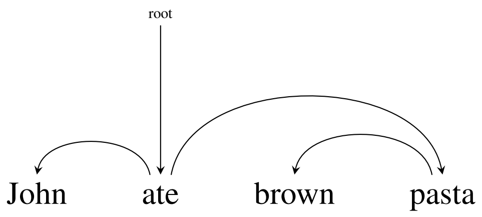


Figure 3.4: A dependency graph for the sentence *John ate brown pasta*.

[1]It is a common practice to treat *to* as a subject-raising auxiliary verb which may also share its subject with its infinitive. See [Hudson, 1976] for more discussion about this analysis.

3.1.1.4 Projectivity

The notion of projectivity is a controversial one. It is used as a common constraint on dependency representation of natural language sentences. Projective dependency trees are restricted to having direct and non-overlapping arcs between nodes, and thus parsers are restricted to output trees similar to the one shown in Figure 3.5 for the sentence *John ate brown pasta which was uncooked*.

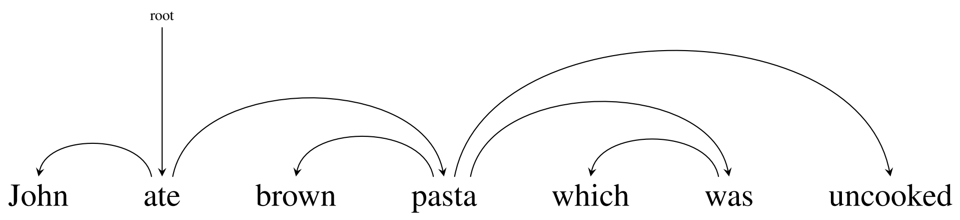


Figure 3.5: Projective dependency tree for the sentence *John ate brown pasta which was uncooked*.

However, in certain situations – especially in languages with more word order flexibility such as Arabic, German, and Russian, it may not be possible to draw projective trees for sentences without overlapping arcs in the same tree, Hence, most dependency variants allow non-projective representation of sentences in order to be able to capture non-local dependencies. For example, if an adverb is included in the sentence *John ate brown pasta yesterday which was uncooked*, the adverb *yesterday* separates the relative clause *which was uncooked* with the verb's object *brown pasta* that it modifies, which results in an overlapping arc in the tree representation as shown in Figure 3.6.

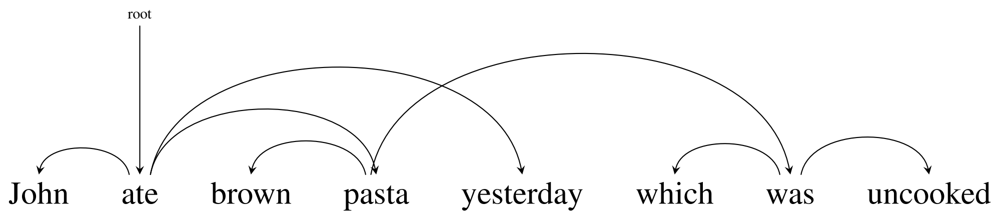


Figure 3.6: Non-projective dependency graph for the sentence *John ate brown pasta yesterday which was uncooked*.

3.1.2 Dependency parsers

In this section, we will give an overview of two state-of-art parsers that are based on dependency framework: (i) MSTParser [MacDonald, 2006], which is a graph-based parser using a bottom- up chart-based parsing strategy, and (ii) MaltParser [Nivre, 2006], which is a transition-based parser using shift-reduce parsing algorithm.

3.1.2.1 Maximum spanning trees parser (MSTParser)

MSTParser is a graph-based parsing system used for identifying maximum spanning trees from a dense graph-based representation of a given sentence [Kübler et al., 2009]. This system uses a machine learning algorithm and parsing algorithms. The machine learning algorithm is used during parser training in order to assign different weights to correct and incorrect trees for a given sentence, whereby higher weights are assigned to correct trees than to incorrect trees. The parsing algorithm is based on a bottom-up chart parsing strategy, where trees are generated and used for searching and finding the highest weighted dependency tree for a given sentence.

During training, the learning algorithm is used for generating a margin between the correct and the incorrect dependency graphs. The more errors a graph has the further away its score will be from the score of the correct graph. During parsing, the parsing algorithm is used for generating dependency graphs [MacDonald, 2006, p. 20], while the generated graph that matches the graphs with the highest weight is selected as the correct analyses.

MSTParser processes projective dependency graphs using a form of projective dependency parsing which is based on Eisner's bottom-up dynamic programming parsing algorithm [MacDonald, 2006]. Eisner's algorithm processes substrings as non-constituent spans (where spans are two or more adjacent words) which are combined to produce larger spans [Eisner, 1996]. A bottom-up dynamic programming algorithm is used to create a dynamic programming table that stores the score of the best subtree, which is the score of two complete subtrees. For example, C[s][t][d][c] represents a dynamic programming table in MSTParser, and stores the score of the best subtree from position s to position t where $s \leqslant t$, d specifies the direction in which the parser may gather left or right dependents, and c determines the completeness of subtrees which takes one of two values for each subtree; 0 for incomplete subtrees and 1 for complete subtrees. For instance, C[s][t][$\leftarrow$][1] would be the score of a complete subtree where all dependents are gathered in rightward fashion, while C[*s*][*t*][←][0] indicates an incomplete subtree.

Some natural languages, such as Arabic, German or Czech, have a more flexible word order than English and require some arcs to overlap with other arcs; i.e., non-projective parsing. Thus, MSTParser uses Chu-Lui-Edmond algorithm [Edmonds, 1967] for generating non-projective graphs by allowing arc crossings [MacDonald, 2006, p. 37]. MSTParser implements non-projectivity by searching the entire space of spanning trees with no restrictions in order to identify the highest scoring graph [MacDonald, 2006, p. 37].

Since this parser is trained on annotated data that is designed for specific languages, it is a language independent parser. It has been applied to 14 diverse languages: Arabic, Bulgarian, Chinese, Czech, Danish, Dutch, English, German, Japanese, Portuguese, Slovene, Spanish, Swedish and Turkish. Additionally, MSTParser is claimed to provide a state-of-the-art accuracy and to be efficient [McDonald et al., 2005].

3.1.2.2 MaltParser

MaltParser is a transition-based parser using a shift-reduce parsing strategy. MaltParser is a data-driven parser trained on annotated treebank that is specifically created to process the syntax of the language in question [Han et al., 2009, Nivre et al., 2006]. This parser is language-independent [Nivre et al., 2006]. It has been successfully applied to a number of different languages such as English, Swedish, Bulgarian, and Arabic [Bengoetxea and Gojenola, 2010]. MaltParser contains a family of algorithms, which are all based on shift-reduce parsing, such as arc-standard, arc-eager, list-based etc., we discuss some of these algorithms in more details in Section 5.2.1. We have based our parser implementation on the arc-standard algorithm of this parser.

There are three main components in MaltParser, these are:

1. Parser: The parser uses a shift-reduce strategy to build a labelled dependency graph in one left-to-right pass over the input. Two main data-structures are manipulated by MaltParser: (i) a stack of partially processed words from a given sentence, and (ii) a queue to store the remaining unprocessed words from the input sentence. Generally, the parser performs one of three basic elementary actions during parsing [Nivre et al., 2006], these actions are:

   SHIFT: Push the first item from the queue onto the stack.

LEFT-ARC(*r*): Add an arc labelled *r* from the first item on the queue to the item on the top of the stack; pop the stack.

RIGHT-ARC(*r*): Add an arc labelled *r* from the item on the top of stack to the first item on the queue; pop the queue.

MaltParser depends on the recommendation of an oracle, which learns different parse actions from a set of annotated data by using a machine learning classifier, to perform SHIFT, LEFT-ARC(r) or RIGHT-ARC(r). Since our parser is largely based on MaltParser and we have described this parser in some detail, we present the pseudo-code of it in Figure 3.7.

```
1. Given an oracle and a string of input text, set the queue
   to the input text and the stack to be empty.
2. Until the queue is empty and the stack has exactly one entry,
   ask the oracle for an action.
   2.a). If the oracle suggests LEFT-ARC(r) then make the head of
         the queue a parent of the  topmost item on the stack and
         pop the stack.
   2.b). If the oracle suggests RIGHT-ARC(r) then make the topmost
         item on the stack the parent of the head of the queue and
         pop the queue.
   2.c). If the oracle suggests a SHIFT then move the head of the
         queue to the top of the stack.
```

Figure 3.7: Pseudo-code for MaltParser.

2. Guide: MaltParser uses history-based parsing models for predicting the next parser action. It uses features of the partially built dependency structure together with features of the (tagged) input string. The parser uses history-based parsing models and discriminative machine learning for determining the best action at each parsing step [Bengoetxea and Gojenola, 2010].

3. Learner: A machine learning algorithm is used in MaltParser to induce a mapping from parser histories, relative to a given feature model, to parser actions, relative to a given parsing algorithm. One of two machine learning algorithms could be used in MaltParser:

   (a) Memory-based learning and classification: MaltParser stores all training instances at learning time and uses some classifications to predict the next action at parsing time. MaltParser uses the software package TiMBL to implement this learning algorithm [Daelemans et al., 2003].

(b) Support vector machines rely on kernel functions to induce a classifier at learning time, which can be used to predict the next action at parsing time. MaltParser uses the library LiBSVM to implement this algorithm with all the options provided by this library [Chang and Lin, 2011].

MaltParser is characterized as a deterministic transition-based parser [Øvrelid et al., 2009, Nivre et al., 2006]. The parser produces dependency trees by transitioning through abstract state machines. It learns models that predict the next state given the current state of the parser, a history of parsing decisions and the input sentence [Kübler et al., 2009]. The parser starts in an initial state, then moves on to subsequent states – based on the prediction of the history-based feature models - until a termination state is reached. Transition based parsing is considered highly efficient, with run-time often linear with the input string length. Furthermore, transition-based parsers can easily incorporate arbitrary non-local features because the current parse structure is fixed by the state. However, errors may propagate in this parser if early incorrect predictions are made [Nivre et al., 2010].

MaltParser has two different modes of operations:

Learning mode: This uses a (training) set of sentences with dependency graph annotations such as input, derives training data by reconstructing the correct transition sequences, and trains a classifier on this data set, according to the specifications of the user.

Parsing mode: In this mode, the parser uses a (test) set of sentences and a previously trained classifier as input and parses the sentences, using the classifier as a guide.

MaltParser relies completely on induction learning from a treebank for the analysis of new sentences and on deterministic parsing for disambiguation. This combination of methods guarantees that the parser is both robust, producing an analysis for every input sentence, and efficient by deriving analyses in time that is linear in the length of the sentence (depending on the particular algorithm used).

3.2 Phrase structure framework

Another significant framework for representing natural language syntax is the Phrase Structure framework (PSF), which was initially formulated by Bloomfield [Bloomfield, 1933] for modelling English syntax. Chomsky popularised this method in the late 1950s. In PSF, natural languages are described by breaking a sentence down into its constituent parts. There are a number of variations that have been derived from this type of framework. In this section, we will describe some of the theoretical notions behind this framework which are implemented in some of the variations of the framework.

3.2.1 Theoretical background

3.2.1.1 Derivational versus transformational rules

Rewrite rules are used for deriving sentence structures from rules by using atomic labels for nodes [Chomsky, 1959], e.g., $X \rightarrow Y$ rule means rewrite X as Y. A set of rewrite rules can be used for describing various types of sentences in a given language. The rules in Figure 3.8 show that a sentence *S* can be derived from a Noun Phrase (*NP*) followed by a Verb Phrase (*VP*), and an *NP* is derived from a Determiner (*DET*) followed by a Noun (*N*), and the actual words *a* can be related with *DET*, *boy* with *N*, *runs* with *V* and so on. Thus, according to these rules, sentences such as *the boy runs* and *the dog walks* are well-formed and correct sentences.

```
1. S   →  NP  VP
2. NP  →  Det  N
3. VP  →  V
4. DET →  the
5. DET →  a
6. N   →  boy
7. N   →  dog
8. V   →  walks
9. V   →  runs
```

Figure 3.8: Context-free phrase structure grammar rules.

However, Chomsky [Chomsky, 1957, p. 57] claims that applying derivational rules, as in Figure 3.8, for defining the relationships between elements of non-simple sentences is complicated and difficult to achieve. Alternatively, transformational rules are used for describing sentences.

By using transformational rules, we can analyse a sentence that is a derivation from the structure of some other structurally related sentences [Liles, 1971, p. 43]. For example, the sentence in 5(b) is the passive form of the sentence in 5(a).

(5)

(a) *The judge dismissed the case*

(b) *The case was dismissed by the judge*

The passive transformational rule, as shown below, involves switching and inserting items. The subject (*the judge*) and the object (*the case*) of the noun-phrases are switched. Then, *by* is inserted infront of the original subject noun-phrase which now follows the main verb (*dismiss*), and *be+tense* (*was*) is added to the auxiliary.

$\mathtt{NP_1 + tense + Aux + V + NP_2 + X} \rightarrow \mathtt{NP_2 + tense + Aux + V + by + NP_1 + X}$

Since the passive transformational rule ought to only be applied to sentences containing 'action' verbs[2], it cannot be applied to sentences with 'description' verbs[3]. Thus, passive transformation rules can produce ungrammatical sentences in the following way:

```
John is a fool → a fool is been by John
```

This limitation exists because the passive focus is shifted from the active elements to those being acted upon. Since the actor/acted upon relationship does not exist in sentences with description verbs, the passive relation does not exist either [Lester, 1971, p. 131].

Moreover, it is difficult to deal with linguistic phenomena, such as agreements, in a completely satisfactory way using transformational rules [Pareschi and Miller, 1990]. The notion of feature structures for describing linguistic objects is treated in some other variations of phrase structure framework, such as feature based informational elements [Vijay-Shanker and Joshi, 1988].

[2]verbs that take object noun-phrases.

[3]verbs that take predicate nominal noun-phrase.

3.2.1.2 Features

The organisation of features within a syntactic analysis varies between different feature-based formalisms but, in most of them feature sets are represented as attribute-value matrices (AVMs). For example, Lexical Functional Grammar formalism (LFG) [Bresnan and Kaplan, 1982], includes elements of both constituent structure (c-structure), and functional structure (f-structure).

C-structures are organised using X-bar theory [Bauer, 2001, Kornai and Pullum, 1990]. For instance, the basic functional category *I* is the head of *I'*. As such, an *NP* is headed by an *N*, and an *IP* is headed by an *I* in Figure 3.9.

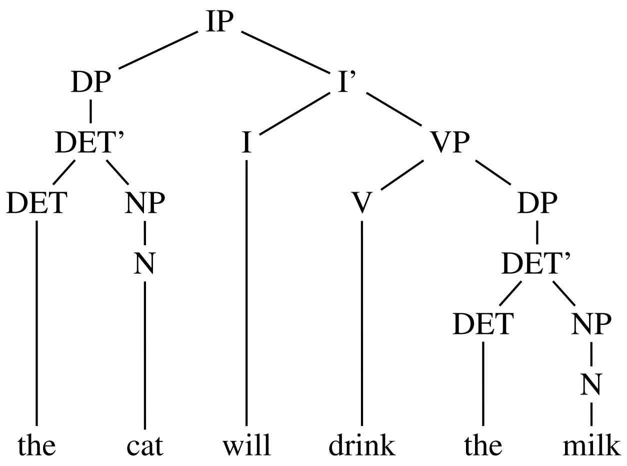


Figure 3.9: LFG c−structure for the sentence *the cat will drink the milk.*

The f-structures represent grammatical functions such as subject and object, and which take the form of sets of paired attributes and values represented in an AVMs [Dalrymple et al., 1995, p. 84] such as TENSE or NUMBER, or grammatical functions such as SUBJECT and OBJECT. An f-structure for the sentence *the cat will drink the milk* is shown in Figure 3.10.

From Figure 3.10, we can recognise three layers of f-structures: the outer f-structure (which corresponds to the whole sentence) has four attributed names: SBJ, TENSE, PRED and OBJ. The TENSE and PRED attributes have simple values (FUT and 'drink <SUBJ, OBJ>' respectively), while SUBJ and OBJ attributes are functions that contain subordinate f-structures as their values. The inner f-structures also have attributes and values, the SBJ takes DEF and PRED attributes and it assigns $+$ and 'cat' as their values respectively, while the OBJ function takes three attributes (DEF, PRED, and NUM) with values such as $+$, 'milk' and SG respectively; the third layer of the f-structure is the values of the attributes such as DEF, PRED and NUM. In this way, LFG represents

$$\begin{bmatrix} \text{SBJ} & \begin{bmatrix} \text{DEF} & + \\ \text{PRED} & \text{`cat'} \\ \text{NUM} & \text{SG} \end{bmatrix} \\ \text{TENSE} & \text{FUT} \\ \text{PRED} & \text{`drink}\left\langle \text{SUB, OBJ} \right\rangle\text{'} \\ \text{OBJ} & \begin{bmatrix} \text{DEF} & + \\ \text{PRED} & \text{`milk'} \\ \text{NUM} & \text{SG} \end{bmatrix} \end{bmatrix}$$

Figure 3.10: LFG f−structure for the sentence *the cat will drink the milk.*

all the functional information of the whole sentence.

Another significant feature-based formalism is Head-driven Phrase Structure Grammar (HPSG), which is a heavily lexicalised formalism, which relocates linguistic information from phrase structure rules into the lexicons [Flickinger et al., 1985].

One of the main features of HPSG is that it contains a set of universal and language specific principles [Kepser, 2000]. For instance, the lexical head is the most important element of a phrase because it incorporates the syntactic information (such as part-of-speech and dependency relations with their constituents) and the semantic information [Pollard and Sag, 1994]. Flickinger et al., [Flickinger et al., 1985, p. 168] state that heads in HPSG refer to linguistic forms (words and phrases). These forms exert syntactic and semantic restrictions on the phrases to form larger phrases. Therefore, verbs are regarded as the heads of verb phrases, nouns as the heads of noun-phrases, and so forth. Additionally, signs are used as a formal representation of combinations of phonological forms and syntactic/semantic structures which can be used for expressing which phonological form signifies a particular syntactic/semantic structure [Sag and Wasow, 1999, p. 356]. These are used for denoting heads and representing the typed feature structures [Miyao and Tsujii, 2008]. They are then used for generating strings, which are defined by their location within a type hierarchy and by their internal feature structure represented by AVMs [Sag and Wasow, 1999, Pollard and Sag, 1994] as shown in Figure 3.11 for the word *walks* [4]

[4] A sign of the type word with a head of the type verb *walks* has no complement but requires a subject that is a third person singular noun.

$$\begin{bmatrix} \textit{word} & \\ \text{PHONE} & \text{‘}\left\langle \textit{walks} \right\rangle\text{’} \\ \begin{bmatrix} \text{HEAD} & \textit{verb} \\ \text{SBJ} & \begin{bmatrix} \text{SYNSEM} & \\ \text{HEAD } \textit{noun} & \\ \text{CONT} & \begin{bmatrix} \text{PER } \textit{3rd} \\ \text{NUM } \textit{sing} \end{bmatrix} \\ \text{COMP} & \end{bmatrix} \end{bmatrix} & \end{bmatrix}$$

Figure 3.11: HPSG structure for the word *walks*.

3.2.2 Phrase structure based parser

3.2.2.1 Stanford parser

A Stanford parser is a statistical parser, which can be trained on annotated data, and applied to a number of languages such as English, Chinese and Arabic.

There are different model implementations of the Stanford parser, as stated by Klein and Manning [Klein and Manning, 2003].

Socher et al., [Socher et al., 2013] describe the most recent model of this parser[5]. They introduce a Compositional Vector Grammar (CVG), which combines Probabilistic Context-free Grammar (PCFG) with a Recursive Neural Network (RNN) that learns syntactico-semantic, and compositional vector representations.

The CVG helps in capturing the discrete categorisation of phrases into NP or PP while the RNN helps in capturing fine-grained syntactic and compositional-semantic information about phrases and words. The combination of the two helps finding the syntactic structure as well as capturing compositional semantic information, which gives the parser access to rich syntactico-semantic information in the form of distributional word vectors and computes compositional semantic vector representations for longer phrases.

3.2.3 Probabilistic grammars

The frameworks described in the previous sections (phrase structure and dependency based frameworks) can have probabilities assigned to their elements. In this section,

[5]The new model is based on the model described by Klein and Manning [Klein and Manning, 2003].

we will attempt to briefly describe the way probabilities are assigned to grammatical rules.

The focus on data-oriented and corpus-based methods have led to the development of probabilistic grammars. These are considered to be one of the core modelling techniques for processing syntactic structures [Charniak, 1996]. Moreover, probabilistic grammar is considered attractive for different reasons [Corazza and Satta, 2006]. It is amenable to inspection by humans. Thus, it is relatively easy to understand which tendencies the model has captured if the underlying rules are understandable [Collins, 1999]. The frequencies of various structures in the training data are collected in order to analyse the characteristics of the parsing model, which is used for providing a reasoning mechanism in the face of ambiguity that often exists in natural languages [Collins, 2003].

3.2.3.1 Probabilistic context-free grammar (PCFG)

A basic CFG rule consists of 4-tuple G = (N, Σ, P, S): N is a finite set of non-terminal symbols, Σ is a finite set of terminal symbols, P is a finite set of productions or rewrite rules e.g., a P rule takes the form $\alpha \rightarrow \beta$ where $\alpha \in N$ and $\beta \in \{N \cup \Sigma\}$, and $S \in N$ is the starting symbol.

By associating probabilities with the productions of context-free grammar rules, we can obtain a probabilistic context-free grammar (PCFG). Thus, a PCFG is a simple modification of a context-free grammar [Collins, 1999]. For example, we can associate a probability with the rule $\alpha \rightarrow \beta$ as $P(\alpha \rightarrow \beta \mid \alpha)$. We can then use this as the conditional probability of choosing the rule $\alpha \rightarrow \beta$ where α is a non-terminal that is rewritten in a derivation D. Using D as a function for associating a probability to each member of P then the PCFG would be a 5-tuple as $G = (N, \Sigma, P, S, D)$.

Corazza and Satta [Corazza and Satta, 2006] state that PCFGs can be used for describing tree-shaped structures of underlying sentences, which may tackle ambiguity problems by assigning a probability to each parse tree, thereby ranking competing trees in order of plausibility. PCFG contributes to some of the state-of-the-art statistical parsers. Charniak et al., [Charniak et al., 1998] propose a method for counting the number of times that a context-free rule is used in a corpus containing parsed sentences, so that the probability of a particular production rule being used is equal to the number of times that a particular rule is used during training, divided by the total number of times that all rules with the same non-terminals on the left hand side are used during training. We will use similar technique to assign probabilities to constraint rules in

Section 6.5.4

3.2.4 Statistical parsers

3.2.4.1 Robust accurate statistical parser (RASP)

RASP [Briscoe et al., 2006] is a statistical parser that uses raw texts as inputs and outputs different parse analyses, such as parse trees, grammatical relations between lexical heads, or minimal recursion semantic analyses. Unlike the previously described parsers, this parser has a modular pipeline architecture whereby one module feeds the next one. These modules are tokenisation, parts-of-speech (POS) and punctuation tagging, morphological analyser, parser, and a statistical disambiguator. A graphical representation of RASP architecture is shown in Figure 3.12:

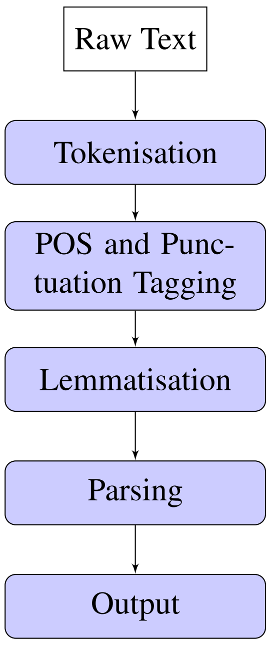


Figure 3.12: RASP pipeline.

Annotation: This parser performs the following annotation tasks on its input:

- Tokenisation: Unlike many other statistical parsers that use annotated data as input, RASP takes a set of raw text or transcribed speech as input. This module tokenises the input by using spaces for separating words and punctuation and marking the boundary of sentences. The system uses a set of deterministic finite-state rules for tokenising the input. The tokenised items are then supplied to the POS and Punctuation tagging module for further processing.

- POS and Punctuation tagging: A first-order ('bigram') Hidden Markov Model (HMM) based tagger is trained on a manually-corrected tagged version of three corpora: the Susanne corpus [Sampson, 2015], the Lancaster/Oslo-Bergen Corpus (LOB) [Johnasson et al., 1978], and a subset of the British National Corpus (BNC) [Burnard, 2000]. POS and punctuation labels are then assigned to the tokenised items. The tagged tokens are then processed to retrieve a token's lemma with any inflectional affixes using the morphological analyser module.

- Lemmatisation: For this module, 1400 finite-state rules are compiled into an efficient C program. This finite-state transducer is applied to the threshold output from the POS and punctuation tagging module (token-tag pair) to deterministically retrieve a lemma with any inflectional affixes.

POS and punctuation sequence parsing: The lemmas plus the affixes are supplied to a modified version of a probabilistic generalised LR parser. This parser utilises 689 manually-developed and wide-coverage unification-based phrase structure rules [Briscoe et al., 2006]. These rules are automatically converted into an atomic categorised context-free grammar which is used to construct a non-deterministic LALR table. The parser is then guided by this table to perform parse actions which result in the building of a packed parse forest. The associated probabilities that are associated with specific parse actions in the probabilistic *LR* table are assigned to sub-analyses in the forest. The most probable parses are then extracted from the forest by unpacking sub-analyses, following pointers to contained sub-analyses and choosing alternatives in order of probabilistic ranking.

Parser outputs: The parser can produce three types of output: it can output syntactic trees, and/or (weighted) grammatical relations between lexical heads, or produce a minimal recursive semantic representation consisting of a sequence of elementary predictions and possibly underspecified equational constraints.

Briscoe and Carroll [Briscoe and Carroll, 2006] report on the accuracy of RASP using three figures for precision, recall, and F1 score with relation to the grammatical relations in the testing data, which are taken from the PARC DepBank [King et al., 2003], of 81.5%, 78.1%, and 79.7% respectively, according to Briscoe and Carroll [Briscoe and Carroll, 2002]. The other published scores vary between 80-84%.

3.2.4.2 Maximum entropy inspired parser (MaxEnt)

MaxEnt is a statistical parser trained on annotated data, using a machine learning technique, which constructs parse trees by following a sequence of actions similar to standard shift-reduce parse actions [Ratnaparkhi, 1999]. The parser uses different procedures to produce parse actions. These procedures are: TAG, CHUNK, BUILD, and CHECK. They are applied in three left-to-right passes over the input string.

Each action that is produced by the above procedures takes a derivation $d = a_1 ... a_n$ and predicts a new parse action $a_n + 1$ that will result in a new derivation $d^i = a_1 ... a_n + 1$. A maximum entropy probability model is used to assign a score to each action that is produced by the procedures.

A complete derivation tree T is obtained by following a sequence of parse actions, $a_1 ... a_n$. The parser only follows permitted actions when constructing or comparing parse trees, where the actions are scored using a maximum entropy probability model. The maximum entropy models are initially trained on a set of annotated data (Wall Street Journal Treebank [Marcus et al., 1993]) in order to learn derivations of parse trees from it. The score of the complete tree is computed from the score of each action in all the derivation trees that lead to the complete parse tree.

A TAG procedure is used in the first pass for assigning a part-of-speech (POS) tag to each word in the sentence producing a (word, POS tag) pair. In the second pass, the output of the first pass is used for determining the “flat”[6] phrase chunks of the sentence using the CHUNK procedure and assign a “chunk" tag to each (word, POS tag) pair. The chunk tags are then used for chunk detection where the result is a tree forest. Finally, in the third pass, BUILD or CHECK procedures are alternated. A BUILD procedure used to decide whether a tree should be the start of a new constituent or should be merged with an incomplete constituent that is immediately to its left. The most recently produced constituent is then identified by CHECK, which decides whether it is complete. If a constituent is complete then CHECK answers **yes** and adds this to the forest where BUILD processes it, otherwise it answers **no** and BUILD processes the next tree in the forest. This final pass terminates if CHECK identifies a constituent that spans the entire sentence. The **yes** action of CHECK corresponds to the reduce action of standard shift-reduce parsers while **no** corresponds to the shift action. However, the difference between shift and reduce action in shift-reduce parsers and BUILD and CHECK is that the former produces a constituent over several small incremental steps while the latter creates a constituent in one step (reduce a).

[6]A phrase that is a constituent but its children are not constituents is considered “flat".

A set of weighted features is implemented in the maximum entropy framework for modelling training data. These features are based on four intuitions relating to the usefulness of using head words, a combination of head words, less-specific information and the allowance of a limited lookahead [Ratnaparkhi, 1999]. A useful subset of features is extracted from the set of all possible features for use in the maximum entropy model corresponding to a procedure X.

An experiment on training the parser on the Wall Street Journal Treebank, sections 2 to 21, (about 40,000 sentences), show that it can achieve 87.5% precision and 86.3% recall when tested on section 23 of the treebank (about 2,416 sentences) [Ratnaparkhi, 1999, p. 30].

3.2.4.3 The Charniak parser

There are different models of the Charniak parser. However, the basic implementation of all of the models is based on chart parsing. The chart parser processes an agenda of items during each parse iteration. Items are taken from the agenda and processed one by one and then added to the chart. If an item is already in the chart then it is discarded. Otherwise it is used for extending and creating additional items. A PCFG is used as a starting point for associating a Figure of Merit (FOM[7]) to each edge in the chart, while the FOMs are used for judging the edges as to which edge is processed first [Charniak et al., 1998].

The most recent model of the parser implements a maximum-entropy inspired model which is a probabilistic generative[8] model, whereby a probability $p(s,t) = p(t)$ is assigned to every sentence s and every parse tree t. This probability equality holds only for any s when the parser returns t, which maximises this probability [Charniak, 2000]. This parser has been tested on the Wall Street Journal, Section 23. It is reported by Charniak [Charniak, 2000] that this parser achieves a labelled precision and labelled accuracy for sentences $\leqslant$ 40 word of 90.1% while the labelled precision and labelled accuracy for sentences with $\leqslant$ 100 words are 89.6% and 89.5% respectively.

[7]Charniak uses the term FOM because the scores he is using are not strictly probabilities and hence to label them as such would be misleading.

[8]Syntactic trees are ‘generated’ by applying grammatical rules in a top-down manner.

3.2.4.4 The Collins parser

Another popular statistical parser is the Collins parser. Like the Charniak parser, this also has a probabilistic generative model. The first probability model to be implemented was a top-down derivation probability model whereby the model assigns a probability $p(s|t)$ to each parse tree t to a sentence, and then the parser searches for the tree T_{best} that maximises $p(t|s)$ [Collins, 1996].

The second model of the parser extends the first model but it emphasises on the distinction between complements and adjuncts. Complements are identified by attaching "-C" suffix to all non-terminals in the training data that satisfy certain conditions and all modifiers are treated as adjuncts.

In order to solve the problem with regard to wh-movement, a third model is implemented by extending the second model by adding a factor for the probability of gaps and filler creation and transmission. The probability of a rule's gap requirement is calculated so that if a gap requirement is in the subcategorisation frame then the probability of a (gap) TRACE constituent being generated is higher, and if a gap requirement is in the subcategorisation frame, then the probability of a (filler) extra constituent is also higher.

Extending the first model has improved the parser's labelled precision and labelled recall by 0.7% and 0.5% respectively (see Collins [Collins, 1997] for a more detailed description of the parser and the result).

3.3 Summary

In this chapter we have described two substantially different frameworks for parsing natural languages: dependency framework and phrase structure frameworks. We have briefly highlighted the theoretical background of both of these frameworks. Also, some of the state-of-the-art parsers that are based on these frameworks are briefly described. Additionally, the application of probabilistic models to natural language parsing is outlined.

In Table 3.1 we present different features of the parsers that we have described in this chapter. Also, we present our parser where we show the features of other parsers that are present in our parser.

Our parser is largely based on MaltParser, which is a dependency data-driven parser (see Section 5.2 for our parser implementation), but we use features from some other parsers presented in Table 3.1. We are using features of chart parsing which is used in

Parsers	Dependency/PST	Grammar-driven/Data-driven	Statistical	Chart
MaltParser	Dependency	Data-driven	No	No
MSTParser	Dependency	Data-driven	No	Yes
Stanford	both	Data-driven	Yes	No
RASP	Dependency	Grammar-driven	Yes	No
MaxEnt	Dependency	Data-driven	Yes	No
The Charniak parser	PST	Data-driven	No	Yes
The Collins parser	PST	Data-driven	Yes	No
Our Parser	Dependency	Both	No	Yes

Table 3.1: A comparison of features of different parsers.

MSTParser (see Section 5.2.3 for the way we used chart parsing). We also integrate different types of constraint rules, such as probabilistic lexicalised local subtrees and probabilistic unlexicalised local subtrees, that are described in Section 6.5.4, where we assign conditional probabilities in the same way as in the Charniak parser. Our parser could be considered as a partially grammar-driven parser because we integrate constraint-rules in it, where these rules influence the decision of our parser in certain situations (see Section 6.5.1 for more details).

Since the experiments conducted in this study are on Arabic, we will describe the structural complexities of this language in the following section, but, firstly, in order to have an idea about the performance of other parsers which have been tested on Arabic; we will present the published parse results for the CoNLL Arabic shared task [Nivre et al., 2007] in Table 3.2[9].

#	Parsers	LAS Accuracy (%)
1	Alabbas	80.40
2	Stanford	77.72
3	MaltParser	76.52
4	Nakagawa	75.08
5	Hall, J.	74.75
6	Sagae	74.71
7	Chen	74.65
8	Titov	74.12
9	Hall, K.	73.40
10	our parser	72.7
11	Attardi	72.66

Table 3.2: Published parse accuracy for Arabic [Nivre et al., 2007].

[9]The names of the parsers in this table are as given by Nivre et al., [Nivre et al., 2007]

We present the accuracy measure of the parsers in terms of labelled attachment scores (which is based on correct dependency relations between two tokens with correct grammatical functions (labels)). It is worth noting the results of these parsers cannot be compared to the results we obtain in Chapter 6 because most of these parsers are trained and tested on a different treebank (Prague Arabic dependency treebank (PADT) [Smrž et al., 2008] while our parser is tested on the Penn Arabic treebank (PATB) [Maamouri and Bies, 2004]. However, the result published by Alabbas and Ramsay [Alabbas and Ramsay, 2011][10], which is the highest of all previously published results, is based on using a combination of the output of MaltParser [Nivre, 2008] and MSTParser [MacDonald, 2006].

[10]The result for this parser is based on the PATB rather than PADT which was used for testing the parser in the CoNLL shared task.

Chapter 4

The Challenges of Parsing Arabic

The complex structure of Arabic creates a large number of ambiguities in the language. Farghaly and Shaalan [Farghaly and Shaalan, 2009] state that the average number of ambiguities per token in Arabic is 19.2 while in most other languages it is 2.3 per token. This high level of ambiguity in Arabic makes parsing it a challenging task [Attia, 2008]. Ambiguity, particularly local ambiguity, is a central problem in natural language parsing. In this chapter, we will briefly highlight some of the structural complexities of Arabic which may make sentences ambiguous.

Since we have conducted all of the experiments in this study on Arabic, and because Arabic is more complex than most other languages such as English, we devote this chapter to briefly describing some of the structural complexities of Arabic, which may affect parsing performance. Chalabi [Chalabi, 2004a], Daimi [Daimi, 2001] and Fehri [Fehri, 1993] all argue that there are many complexities and subtleties in Arabic while Holes [Holes, 2004] states that Arabic has a complex syntactic structure.

Tayli and Al-Salamah [Tayli and Al-Salamah, 1990] state that Arabic alphabet consists of 28 letters that can be extended to a set of 90 by using additional shapes, marks and vowels. As with other natural languages, there are consonants and vowels in Arabic. From the 28 Arabic letters there are three vowels, which are اَ (pronounced as /a:/), اِ (pronounced as /i:/), and اُ (pronounced as /u:/). Some long vowels are constructed in Arabic by combining اَ, اِ and اُ with short vowels. According to Tayli and Al-Salamah [Tayli and Al-Salamah, 1990] and Ramsay and Mansour [Ramsay and Mansour, 2006], short vowels and certain other examples of phonetic information such as the double consonant (shadda) are not represented by letters but by diacritics, which are largely missing in Modern Standard Arabic (MSA).

4.1 Lack of diacritics in MSA

A diacritic symbol in Arabic is a short stroke placed above or below the consonant. Arabic diacritics are divided into three sets: short vowels, doubled case endings, and syllabification marks.

Short vowels: These are written as symbols, either above or below the letter in the text with diacritics. These are:

- fatha: represents the /a/ sound and uses an oblique dash over a letter as in اَ.
- damma: represents the /u/ sound and uses a loop over a letter that resembles the shape of a comma as in اُ.
- kasra: represents the /i/ sound and uses an oblique dash under a letter as in اِ.

Doubled case ending: This is when vowels are used at the end of the words; the term *tanween* is used to express this phenomenon. *Tanween* suggests indefiniteness and is manifested in the form of case marking or in conjunction with case marking as the bearer of *tanween*. Similarly to short vowels, three different diacritics are used for tanween: (i) tanween al-fatha as in اً /aN/, (ii) tanween al-damma as is اٌ /uN/, and (iii) tanween al-kasra as in اٍ /iN/. They are placed on the last letter of the word and have the phonetic effect of placing an 'N' at the end of the word.

Arabic diacritised text also contains two syllabification marks:

- shadda: a gemination mark placed above Arabic letters. It denotes the doubling of the consonant. The shadda is usually combined with a short vowel, as in رَّ.
- sukuun: written as a small circle as in اْ. It marks the boundaries between syllables or ends of verbs in cases using the jussive moods. This indicates that the word does not contain vowels.

Syllabification marks: Arabic texts without diacritics have considerable ambiguities, which may be problematic for NLP applications because many words with different diacritic patterns appear to be identical in a diacriticless setting [Zitouni et al., 2006]. Moreover, diacritising Arabic texts will give clear indication of the meaning of words and their syntactic roles in sentences. For instance, the word علم *alam* can have many meanings and roles when diacritised. It can be a noun, as in عِلْمٌ *'ilmuN* "knowledge". In other cases such as عَلَمٌ *'alamuN* "flag", it can act as a passive transitive verb as in عُلِّمَ *'u-llima* "is taught" or as active transitive verb as in عَلِّمْ *'a-llim* "teach" and عَلَّمَ *'a-llama* "taught". Also, It can act as passive intransitive verb as in عُلِمَ *'ulima* "is known" or it can act as active intransitive verb as in عَلِمَ *'alima* "knew". These examples indicate that single words may have many roles when diacritised, and that they can be correctly recognised in diacritised settings. Parsing MSA text is challenging because it is written without diacritics, and as we have discussed in this section the lack of diacritics in MSA text makes parsing a challenging task.

4.2 Free word order

It is pointed out in the related literature that Arabic has a high syntactic flexibility [Daimi, 2001]. One of the sources of ambiguities in Arabic is its relatively free word order [Attia, 2008, p. 179]. The canonical order of an Arabic sentence is verb-subject-object (VSO). However, a range of other word orders such as verb-object-subject (VOS) order, subject-verb-object (SVO) order and object-verb-subject order (OVS) are also possible [Ramsay and Mansour, 2006]. This leads to considerable ambiguities. Arabic allows the construction of sentences in various orders in which the functions of some words change according to their positions within the sentence.

If the subject and the object(s) follow a verb and there is no visible case marking then it is difficult to decide which order it is (VSO or VOS) i.e., it is difficult to identify the subject and the object. For example, in a sentence such as قاتل الجندي الإرهابي *qAtl aljndy al-'irhAby* "" it is unclear whether الجندي *aljndy* "the soldier" is the subject or الإرهابي *al-'irhAby* "the terrorist" is the subject. The identification of the subject and object of the sentence is based on the pronunciation of the sentence which gives it two different meanings: (i) "The soldier fought the terrorist" where "the soldier" is the subject, and (ii) "The killer of the soldier is the terrorist" where الإرهابي *al-'irhAby* "the terrorist" is the subject.

However, in the SVO order as in the sentence الجندي قاتل الإرهابي *aljndy qAtl al-'irhAby* "The soldier fought the terrorist" it is clear that الجندي *aljndy* "the soldier" is the subject in the sentence and الإرهابي *al-'irhAby* "the terrorist" is the object.

4.3 Zero copula

The zero copula is where there is a relationship between the subject of a sentence and a predicate without an overt marking of this relationship. This phenomenon exists in Arabic and in many other languages. A nominal sentence in Arabic could be formed with a subject noun-phrase (NP) and a predicate [Ramsay and Mansour, 2006], such as كُرَةٌ فِي المَلعَبِ *kuraTuN fy Almal'abi* "a ball on the pitch". However, because there is a degree of word order freedom in Arabic, if the subjects in Arabic sentences are indefinite and the predicates are a preposition or an existential adverb, then the order of the subjects and the predicates is inverted, which allows the predicate to precede the subject, as in فِي المَلعَبِ كُرَةٌ *fy Almal'abi kuraTuN* "on the pitch is a ball".

4.4 Arabic clitics

Clitics are defined as morphemes that possess the syntactic characteristics of a word. However, they are morphologically bound to other words [Crystal, 1980]. It is possible to attach clitics to the beginning or to the end of Arabic words and this almost always alters their formation and meaning. Clitics can often alter word types from nouns to verbs, or even change the verb type from transitive to intransitive [Nelken and Shieber, 2005]. Hence, clitics can be considered as another source of ambiguity in Arabic.

Conjunctions in Arabic can often appear as clitics and modify Arabic verb, which can cause ambiguities in Arabic sentences, hence imposes challenges to parsers. For example, the sentence in (6) indicates that the noun وليهم *walyahum* "their leader" is ambiguous because the first two letters و /w/ and ل /l/ could be clitics attached to the word يهم *yahum* "take charge". If they are clitics then the word يهم *yahum* "take charge" is a verb, as in (7).

(6) وليهم أحمد في المسألة
wliyahum '.hmdu fy Almasa'lT
their leader Ahmed in the situation (Noun. singular. masculine)
"Ahmed is their leader in the situation"

(7) وليهم أحمد في المسألة
w li yahum '.hmdu fy Almasa'lT
and for to take charge Ahmed of the situation (Verb. singular. third person)
"and for Ahmed to take charge of the situation"

The examples in (6) and (7) indicate that attaching clitics to words makes it difficult to identify not only their meanings but also their syntactic roles in a sentence; i.e., the words become ambiguous and parsers may have to search for all the possible analyses of the same word, which is likely to affect their efficiency.

4.5 Noun multi-functionality

As Arabic nouns encompass a wide range of categories, they are difficult to define in comparison to verbs. One of the reasons that nouns create ambiguities is that some Arabic nouns are derived from verbs, and they can function as verbs in some sentences [Attia, 2008]. For example, البحث *Alba.h_t* "the search/the research" is a noun but it can have two functions: (i) as a noun, as in (8) or (ii) as if it is a verb, as in (9):

(8) استمرّ الباحث في البحث للجامعة
istama-rra AlbA.h_tu fy Alba.h_ti liljAmi'aT
continued the researcher in the research for the university
"the researcher continued in the research for the university"

(9) استمرّ الباحث في البحث عن الجامعة
istama-rra AlbA.h_tu fy Alba.h_ti 'an AljAmi'aT
continued the researcher to look for the university
"the researcher continued to look for the university"

4.6 Arabic pro-drop

Baptista [Baptista, 1995] and Chomsky [Chomsky, 1981] both state that in pro-drop theory, the subject of a sentence can be omitted if the agreement features of the verb are rich enough to recover its content. Attia [Attia, 2008] states that Arabic verbs can recover missing subjects by conjugating themselves to indicate the gender, number and person of the dropped pronominal subject. Thus, in some Arabic sentences, the subject pronoun becomes redundant when the verb can recover it [Farghaly and Shaalan, 2009]. As in the sentence in (10), the verb أكلت *'kalat* "ate" indicates that the missing subject is a singular, feminine and third person pronoun, hence it is omitted.

(10) أكلت الدجاجة *'kalat Al dajAjT* "ate the chicken"

Pro-drops pose two challenges to parsers: (i) to identify whether there is an omitted pronoun or not, and (ii) to identify the antecedent of an omitted pronoun [Chalabi, 2004b].

The challenge of identifying a pro-drop is further increased when dealing with Arabic verbs because they can often be both transitive and intransitive. For example, it is not clear in (10) whether the *NP* الدجاجة *Al dajAjT* "the chicken" following the verb أكلت *'kalat* "ate" is the subject or not. If the *NP* is the subject of the verb *V*, then the sentence would mean *the chicken ate*, which makes the verb أكلت *'kalat* "ate" intransitive, as in Figure 4.1(1). However, if the *NP* is the object of the verb *V* and the subject is an omitted pronoun (as "she") then the sentence would mean *she ate the chicken*, which makes the verb أكلت *'kalat* "ate" a transitive verb as in Figure 4.1(2). Or, it could mean *the chicken was eaten*, which makes the verb أكلت *'kalat* "ate" a passive transitive verb, as in Figure 4.1(3). The ambiguities caused by pro-dropping can lead to three different types of structural analysis of the sentence in (10), as shown in Figure 4.1

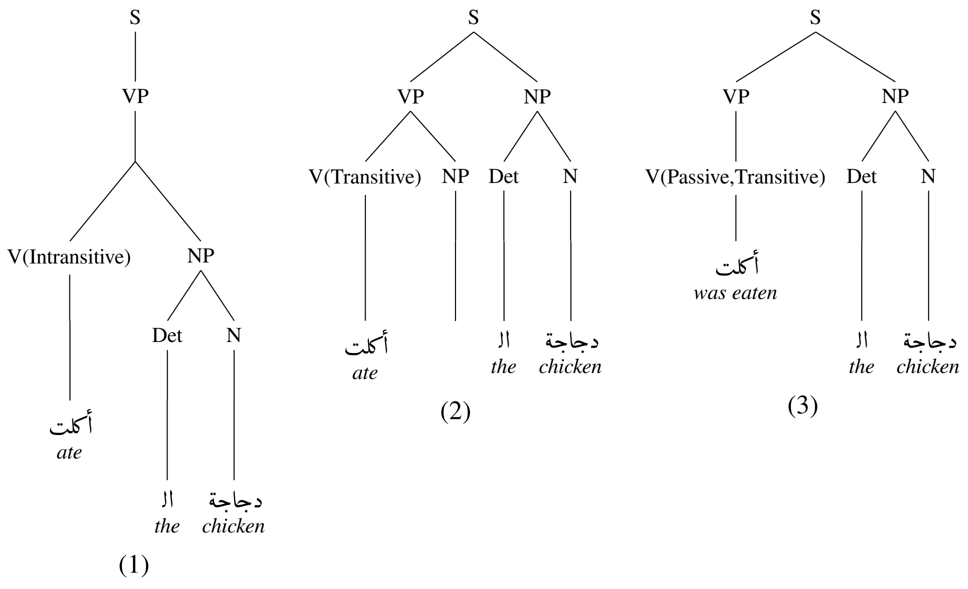


Figure 4.1: Ambiguity caused by pro-drop.

4.7 Summary

The focus of this chapter has been to identify and highlight some of the structural complexities of Arabic which may affect parsing performance. The level of ambiguity in Arabic is the major factor that affects parsers. Here we have focused on highlighting the features of the language that make a sentence ambiguous.

The individual source of structural ambiguities, which often exist in other languages, may not affect parsing performance in isolation. However, when they are combined they can be problematic. For example, English text in a diacriticless setting may not be very problematic but because of the high degree of Arabic word order freedom, a lack of diacritics in Arabic text may create problems for parsing, as we have shown in Section 4.2. Moreover, the zero-copula in Arabic may not be problematic on its own but when it is aggravated with word order freedom, as explained in Section 4.3 the position of subjects and predicates in sentences are inverted (allowing the predicate to precede the subject) when there is a zero-copula.

Chapter 5

Parser Development

Dependency parsing has achieved an impressive level of efficiency and robustness in recent years. This improved performance has contributed to a number of NLP applications such as Machine Translation, Question Answering, Information Extraction and many more [Kübler et al., 2009]. There are two main possible factors that contribute to such improved performance of dependency parsing, which are:

1. The possibility of implementing an efficient and robust parser with a high level of accuracy.

2. The possibility of concealing the differences in surface word order between languages by concentrating on the functional relations between words in dependency parsing, where these relations are largely similar across all languages. Thus, dependency parsers can be used for uniformly treating a number of typologically different languages.

We are, therefore, motivated to base our parser implementation on the dependency framework. In this framework, there are two different approaches to dependency parsing. One approach is data-driven, while the other is grammar-driven. In data-driven parsing, parsers are trained on a set of annotated data, such as treebanks, and a parse model is generated which is used for guidance in parsing unseen sentences. In grammar-driven parsing, parsers are guided by a set of specifically defined grammatical rules for a given language which are used for parsing unseen sentences.

Since a large number of treebanks are available for many natural languages, and they can be seen to have contributed to the development of a number of state-of-the-art data-driven parsers (such as MaltParser and MSTParser), one can develop a language independent data-driven parser without the need for linguistic expertise, or specifically defined grammatical rules for a given natural language. Moreover, since all treebanks have an underlying grammar that has been used by annotators during treebank building, it should be possible to extract constraint rules or patterns from them. In Section 6.5.2 we present different ways of extracting constraint rules from a treebank and integrating them into a data-driven parser.

The focus of this chapter is on describing the implementation of a data-driven parser, which is largely based on the arc-standard algorithm of MaltParser [Nivre, 2008] but with some modifications. The parser implementation is based on a supervised approach, where the input sentences to a machine learning classifier are annotated with correct dependency representations. There are two tasks in supervised data-driven parsing: (i) a learning task, where a set of annotated input sentences S is used for generating a parsing model M, and (ii) a parsing task, where for a set of sentences $S_i = S_1, S_2, \ldots, S_n$, the parser derives a dependency graph G_i for S_i by using M.

After implementing the data-driven parser, different types of constraint rules are extracted from a set of dependency trees that we obtain by converting the Penn Arabic Treebank (PATB) [Maamouri and Bies, 2004] to the dependency format. The effects of constraint rules on parsing performance are demonstrated during the evaluation of the parser in Section 6.5. Our aim in integrating constraint rules into a data-driven parser is to improve parsing accuracy while retaining the benefits of data-driven parsing (in terms of speed and robustness). In order to control the effect of constraint rules on the parser, a scoring technique is implemented whereby the score given to constraint rules determines whether to apply them to the parser or not. In this way we can trade-off between the speed and accuracy of the parser.

In the sections which follow, we describe the data that we are using for training and testing a state-of-the-art parser (MaltParser) and also for training and testing our parser. We then describe an approach for developing a non-deterministic data-driven parsing algorithm which we evaluate by examining Arabic sentences. Additionally, we present an approach for generating a parse model M in Section 5.2.4 from a set of training data for testing on new sentences. The final sections of this chapter are devoted to describing an approach for extracting different types of constraint rules from a treebank and integrating them into the non-deterministic data-driven parser.

5.1 Data for inducing a dependency parser

The kind of data that is suitable for developing a data-driven parser is an annotated treebank. There are a number of treebanks available for inducing a dependency parser for a number of natural languages. Some of the most popular treebanks for Arabic are: Penn Arabic Treebank (PATB) [Maamouri and Bies, 2004], Prague Arabic dependency treebank (PADT) [Smrž et al., 2008, Smrž and Hajic, 2006], and Columbia Arabic treebank (CATiB) [Habash and Roth, 2009a].

Each one of these treebanks is different, particularly in terms of their format and their level of linguistic information. The trees in PATB have phrase structure tree format while the trees in PADT and CATiB have dependency tree format. Moreover, the level of linguistic information in PATB and PADT is complex and very rich (each containing several hundreds of part-of-speech (POS) tags) whereas CATiB encodes less linguistic information and uses a small set of POS tags (six tags only).

All of these Arabic treebanks, in particular PATB and PADT, have been used in a variety of natural language applications, such as semantic role labelling, morphological disambiguation, POS tagging, and parsing. However, these treebanks are not without limitations when they are used for inducing a parse model. The following list is a succinct description of limitations of each treebank for parsing.

- **PATB** employs rich linguistic information such as morphological information, semantic role information and a set of 420 fine grained POS tags. The POS tags specify almost every aspect of Arabic word morphology, including definiteness, gender, number, person, mood, voice and case. There are different versions of PATB (such as PATB part 1 version 2 containing 168,123 tokens after clitic segmentation[1] and PATB part 2 version 2 containing 125,698 tokens[2]) where each version contains various amount of data collected from different Arabic news wires.

  The linguistic information in PATB is sufficient for inducing a parser. However, the limitation for using this treebank directly for generating a parse model is that its annotation schemata is based on a phrase structure format, which cannot be used for dependency parsing. It is, however, fairly easy to convert the tree structures of PATB to dependency structures (see Section 5.1.2).

[1]Available at: https://catalog.ldc.upenn.edu/LDC2003T06.
[2]Available at: https://catalog.ldc.upenn.edu/LDC2004T02.

- **PADT** consists of morphologically and analytically annotated data derived from the Arabic Gigaword[3]. In total, it contains 148,000 tokens[4]. It employs more complex morphological information than PATB. For instance, there is a more sophisticated distinction between nominal and adjectival definiteness, number, and gender in this treebank [Habash and Roth, 2009c]. It also employs a very fine-grained set of POS tags. It is possible to convert the fine-grained POS tags to a coarse-grained tags for parser training, and to assign coarse-grained POS tags to unseen data, using currently available Arabic POS taggers, but we do not have the license for using this treebank, therefore we cannot use it in this study.

- **CATiB** consists of 31,319 sorties (trees) collected from a wide range of Arabic news wires such as: Agence France Presse, Xinhua, AlHayat, Al-Asharq Al-Awsat, Al-Quds Al-Arabi, An-Nahar, Al-Ahram and As-Sabah. The total number of tokens in this treebank is over one million [Habash and Roth, 2009a].

  This treebank employs a much smaller linguistic information than the previous two treebanks (PATB and PADT). It uses only six different POS tags, specifically:

  - NOM (for nominals including nouns, pronouns, adjectives and adverbs).
  - PROP (for proper nouns).
  - VRB (for verbs).
  - VRB-PASS (for passive-voice verbs).
  - PRT (for particles such as prepositions or conjunctions).
  - PNX (for punctuation).

Although it is possible to generate a parse model from CATiB trees, we do not have access to one and therefore we cannot use it in this study.

Since we have unrestricted access to the PATB and it is possible to convert phrase structure trees to dependency trees fairly easily, we will use it for parser training and testing. The conversion strategy that we have used is based on the standard phrase structure for dependency transformation algorithms, as described in Section 5.1.2. Before we describe the approach used for converting phrase structure trees to dependency

[3]Available at: https://catalog.ldc.upenn.edu/LDC2003T12.
[4]PADT is available at: http://ufal.mff.cuni.cz/padt/PADT_1.0/index.html.

trees, it is worth addressing two questions in order to lay the ground for converting PATB tree to dependency trees: (i) what is the common element among phrase structure trees and dependency trees? and (ii) how can we use the common elements in the two different types of trees for converting phrase structure trees to dependency trees? We will answer these questions in the following section.

5.1.1 Relationships between phrase structure and dependency trees

In most linguistic theories, such as X-bar theory, the notion of head plays an important role in determining the main properties of a phrase. A head determines the main properties of a phrase. For example, Figure 5.1(1) shows the phrase structure tree for the sentence in (11) where the noun *N* projects a noun-phrase *NP* and makes the phrase an *NP* while the *PREP* projects a prepositional phrase *PP* and makes the phrase a *PP*, and the same applies to the second *NP* in the tree. This shows that heads in phrases determine the phrases' type.

Heads also have an important role in dependency structures. A common way of representing dependency structures for the sentence in (11) is shown in Figure 5.1(2), where the head is a direct parent of its daughter(s) and the head information is explicitly marked in the dependency structures. From these examples, we conclude that heads are a common element in phrase structures and dependency structures.

(11) قاتل الجنود في الحرب *qAtl aljnwd fy al.hrb* "The soldiers fought in the war"

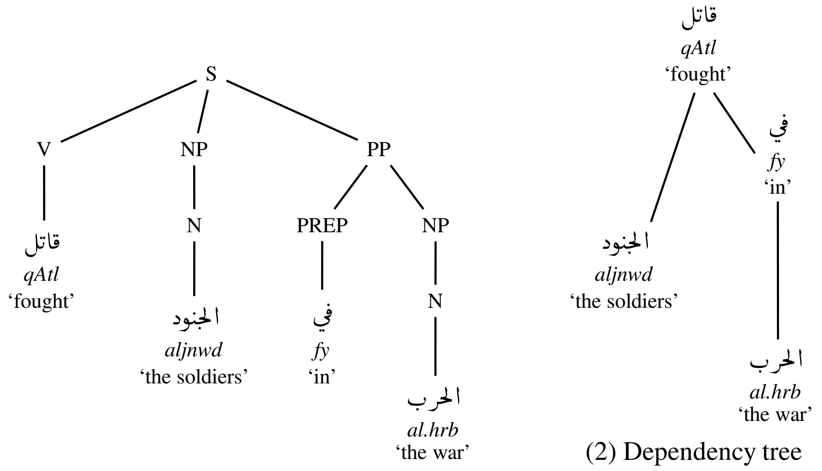


Figure 5.1: Phrase structure and dependency trees for the sentence in (11).

The most widely used method for finding the head of phrase structure trees involves using a head percolation table [Collins, 1997, Magerman, 1995]. Once the head of a phrase structure tree is identified, it is fairly easy to convert the trees in PATB to dependency trees by using the transformation algorithm shown in Figure 5.2, as discussed in detail by Xia and Palmer [Xia and Palmer, 2001]. We will outline the process of converting phrase structure trees to dependency structure trees in the following section.

5.1.2 Conversion of PATB to dependency structures

We use the PATB[5] part 1 version 3 using a 'without-vowel' set of trees for generating a parse model and testing and evaluating our parser. This treebank contains 734 articles written in Modern Standard Arabic (MSA), which were published by the Agence France Presse (AFP) news wire between July and November 2000. The data amounted to 166,068 tokens after clitic segmentation, of which 123,796 tokens were Arabic. There were approximately 5,000 sentences (i.e., 5,000 phrase structure trees) most of which were fairly long. Some of them consisted of more than one hundred words and the average sentence was approximately 29 words. There were some partial sentences in this treebank. The total number of complete sentences with a head category *S* (indicating that the tree is a complete sentence rather than a phrase) as the initial node in the tree was 4,835 while the remainder of the treebank consisted of partial sentences of two or three words length. We ignored these kinds of sentences because they do not contribute positively to parser learning.

The standard conversion algorithm for transforming phrase structure trees into dependency trees is shown in Figure 5.2, this is explained in detail by Xia and Palmer [Xia and Palmer, 2001]. Using the conversion algorithm is a straightforward process. However, the main challenge is to find the subtrees which contain the head in a phrase structure tree.

We can identify the heads of phrase structures by surveying all the phrase structure trees headed by a given label, then we compile a list of labels consisting of the potential labels of the head daughters of the subtrees. This process automatically generates a 'head percolation table (HPT)' [Collins, 1997]. It is called this because lexical items are percolated from their heads to their various projections in a tree. We can see from Figure 5.1(1) that *N* projects to *NP*, which makes the *N* the head-child of the *NP*.

[5]Catalogue number LDC2005T02 from the Linguistic Data Consortium (LDC). Available at: http://www.ldc.upenn.edu/Catalog/catalogEntry.jsp?catalogId=LDC2005T02.

```
1. If the current node is a leaf node then produce a tree with
   no daughter.
2. a). Otherwise, first find the subtree which contains the head
       and convert it into a dependency tree called DEPTREE.
   b). Then, process all other subtrees and convert them into
       dependency trees and add them as daughters of DEPTREE.
```

Figure 5.2: Phrase structure to dependency tree algorithm.

Entries in HPT take the form (x *direction* $y_1/y_2/\ldots/y_n$) where x and y_i are POS tags, *direction* is either left or right (indicating that inspection for the head-child of x should start from left or from right of the set of children of x), and y_i is a set of possible children of x. So, for example, the entry for *S* in the HPT takes the form (*S left V/IV/PV/VERB/VP/NP*). This indicates that the head-child of *S* in the HPT is the first item of the set of head-children of *S* from the left-hand-side of the set *V/IV/PV/VERB/VP/NP*, which in this case is *V*. This interpretation applies to other entries in the HPT such as *NP*, *VP* etc.

In the phrase structure tree in Figure 5.1(1), for example, the root node (*S*) has three children, *V*, *NP*, and *PP*. The conversion algorithm in Figure 5.2 selects the node *V* as the head-child of *S*, because it is the first item from the left-hand-side of the set of children of *S*, and treats the *NP* and the *PP* as the non-head-child. Thus, the head قاتل *qAtl* "fought", which is the head-child of the node *V*, is made the dependency parent of the head الجنود *aljnwd* "the soldiers" and في *fy* "in" of the non-head-child *NP* and *PP* respectively in the dependency structure in Figure 5.1(2).

The items (labels) in the HPT can be ordered in terms of preferences as to which item should be the head-child of a node; i.e., in terms of what we believe to generate a reasonable organisation of dependency trees. Reordering the head-child in the HPT makes the HPT generation process a semi-automatic one[6] and we can then use the algorithm in Figure 5.2 to generate dependency trees by recursively traversing through all the phrase structure trees in the PATB.

It is important to note that the ordering of labels in the HPT table influences the organisation of dependency trees. For example, if the entry in the HPT for the *S* is (*S* left *NP/V/IV/PV/VERB/VP*) instead of (*S* left *V/IV/PV/VERB/VP/NP*), then we will get

[6]We automatically collect a set of children for a head but then we manually reorder the set of children of a head, which we consider a semi-automatic process of creating HPT.

a different dependency tree for the sentence in (11), as shown in Figure 5.3, from that shown in Figure 5.1(2).

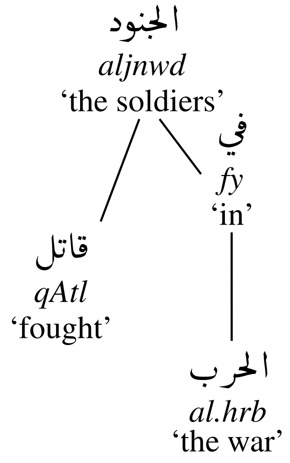


Figure 5.3: A different dependency tree for the sentence in (11) with *NP*s as the head-child of *S*.

Since different organisation of labels in HPT entries produce different dependency trees, it is acceptable to believe that training a parser on trees that are structured in different ways may affect parsing performance. Hence, we experimented with different versions of HPT for converting the PATB trees to dependency trees in order to assess the effectiveness of each version (as shown in Section 5.1.3).

It is important to note that the effectiveness of a particular HPT version on parsing performance does not directly reflect the plausibility of the grammatical structure of the dependency tree produced by that HPT version. In Section 5.1.3.1 we present an approach to transforming a dependency tree produced by a particular version of HPT to a tree that would have been produced by a different version of HPT.

The problems that may arise during the conversion process, which are caused by the nature of the PATB, may include the following:

- **word order freedom and traces:** In order to accommodate the word order freedom in Arabic, traces and indexes are used in PATB for representing subjects and objects of sentences. The function of an item, which indicates whether it is a subject or an object, is used for co-indexing an extraposed item with it. We perform a pre-processing step before converting the PATB from phrase structure format to dependency format by removing traces and replacing them with their co-indexed items. For example, the pronoun هو *hw* "he (3rd.mas.sing.)" in Figure 5.4 in the *NP-TPC-1* phrase where the numeric value *1* is used as its index is co-indexed with a trace in the *NP-SBJ-1* phrase, so that the numeric value of the former phrase matches the index of the latter phrase, which is taken to be the subject of يركب *yrkb* "gets on (3rd.masc.sing)". So, the tree in Figure 5.4 will become identical to the tree in Figure 5.5 when we treat the traces.

```
(...
   (S
      (NP-TPC-1
         (PRON_3MS هو hw "he"))
      (VP
         (IV3MS+IV+IVSUFF_MOOD:I يركب yrkb "gets on")
            (NP-SBJ-1 (-NONE- *T*))
               (NP-OBJ (DET+NOUN+CASE_DEF_ACC الباص AlbA.s "the bus")))) ...)
```

Figure 5.4: An example of a trace in PATB.

```
(...
   (S
      (VP
         (IV3MS+IV+IVSUFF_MOOD:I يركب yrkb "gets on")
            (NP-SBJ-1
              (PRON_3MS هو hw "he"))
               (NP-OBJ (DET+NOUN+CASE_DEF_ACC الباص AlbA.s "the bus")))) ...)
```

Figure 5.5: A transformed version of the tree of Figure 5.4.

Since relations between words are important when representing the dependency structure of a sentence and the fact that traces are not words, any attempt to use them will violate the basic concept of a dependency framework. Therefore, when converting PATB trees to dependency trees, we systematically replace traces with their co-indexed *NP*s.

- **fine-grained POS tags:** PATB employs a rich set of POS tags (420 tags). Some of the linguistic information included in the POS tags consists of inflectional features, such as number and gender agreement, person, and case-marking. Since it is difficult to accurately assign fine-grained POS tags to Arabic using POS taggers (such as MADA [Habash and Roth, 2009b] and AMIRA [Diab, 2009]), it is difficult to accurately parse newly seen data when training data, which is used for parser training, is annotated with fine-grained POS tags.

  Previous experiments conducted by Marton et al., [Marton et al., 2013, Marton et al., 2010] showed that attempting to recognise case-marking hurts parsing accuracy. The negative impact of incorrect case-marking on parsing accuracy with Modern Standard Arabic (MSA) is related to the fact that case-marking agreements do not occur explicitly in undiacritised written MSA thus, parsers cannot adequately learn from them during the training phase, which can subsequently have a negative impact on parsing accuracy.

Since it is difficult to assign fine-grained tags to newly seen data, we reduce the PATB tags to a set of coarse-grained POS tags for parsing Arabic. Thus, when converting the PATB to dependency format we collapse the tag set to a set of coarse-grained POS tags. The simplest way to obtain a set of coarse-grained POS tags from a set of fine-grained POS tags is to remove information from the latter. In Table 5.1 we present an example of a couple of coarse-grained POS tags that we have obtained from a set of fine-grained POS tags. In Table 5.2 we show the set of coarse-grained tags we have extracted from PATB.

Coarse-grained tag	Fine-grained tags
`IV`	`IV1P+IV+IVSUFF_MOOD:I`
	`IV1P+IV+IVSUFF_MOOD:S`
	`IV1P+IV_PASS+IVSUFF_MOOD:S`
	`IV1S+IV+IVSUFF_MOOD:J`
	`IV1S+IV_PASS+IVSUFF_MOOD:S`
	`IV2MP+IV+IVSUFF_SUBJ:MP_MOOD:I`
	`IV2MS+IV+IVSUFF_MOOD:I`
	...
`NOUN`	`NOUN+CASE_DEF_GEN`
	`NOUN+CASE_INDEF_ACC`
	`NOUN+CASE_INDEF_GEN`
	`NOUN+NSUFF_FEM_DU_ACC`
	`NOUN+NSUFF_FEM_DU_ACC_POSS`
	`NOUN+NSUFF_FEM_DU_NOM`
	`NOUN+NSUFF_FEM_DU_NOM_POSS`
	`NOUN+NSUFF_FEM_PL+CASE_DEF_GEN`
	...

Table 5.1: Example of fine-grained and coarse-grained POS tags.

Since it is possible to indicate the functional relations between parents and daughters by using labels, such as *subject* or *object*, in dependency structures, we extract the functional relations from the phrase structure trees that are attached to some of the head phrases in PATB. We use several labels in our dependency trees, which are: COORD (coordinate), DEP (dependent), OBJ (object), PNX (punctuation), ROOT (root), and SBJ (subject).

- **Initial و *w* "and" conjunction:** The conjunction و *w* "and" marks the start of

ABBREV	DET+NOUN	INTERROG_PART	NOUN_NUM	PUNC
ADJ	DET+NUM	IVSUFF_DO	NUM	PV
ADV	DEM_PRON	LATIN	NUMERIC_COMMA	PVSUFF_DO
CONJ	EXCEPT_PART	NEG_PART	PART	RC_PART
CV	FOCUS_PART	NO_FUNC	POSS_PRON	REL_PRON
DET	IV	NOUN	PREP	SUB_CONJ
DET+ADJ	INTERJ	NOUN_PROP	PRON	SUB
VERB	VERB_PART			

Table 5.2: Coarse-grained POS tags extracted from PATB.

large numbers of sentences in PATB. Intuitively, we may think that this conjunction is implicitly conjoining the first clause of the current sentence to a previous sentence, which may encourage us to treat it as its head. However, doing so may conflict with the way in that PATB treats it. In PATB, if و *w* "and" appears at the beginning of a sentence, it is treated as the head of the whole sentence rather than as an item relating the first clause of the current sentence to a previous sentence. For instance, a sentence with a general form such as CONJ S_1, CONJ $S_2 \ldots$CONJ S_n we may bracket it as CONJ (S_1 CONJ $S_2 \ldots$CONJ S_n) but the PATB brackets it as (CONJ S_1) CONJ S_2 CONJ ...CONJ S_n. In order to make a distinction between the sentence-initial conjunction that is treated as the head of the whole sentence with similar conjunctions that conjoin sentence clauses, we will mark the sentence-initial conjunctions as ICONJ. This technique has been used previously by Alabbas [Alabbas, 2013] and has improved parsing accuracy.

- **verbless or equational sentences:** Verbless or equational sentences in Arabic consist of a noun-phrase (subject) and another noun-phrase, a prepositional phrase or an adjectival phrase where they are treated as the predication of sentences. Semantically, we could use the predication as the head of the sentence for treating Arabic verbless sentences, which is arguably the right thing to do [Alabbas, 2013]. However, because our parser uses local contexts as clues to guide its decisions and because previous experiments by Alabbas [Alabbas, 2013] on MaltParser showed that treating the subject as the head yielded more accurate result, we will use the subject of a verbless sentence as the head.

5.1.3 The evaluation of PATB conversion

We have mentioned previously that the ordering of a head-child in terms of preferences in the HPT affects the organisation of dependency trees, where we use these trees for parser training and testing. In order to check the effectiveness of different HPTs, we have compiled different versions of them for obtaining dependency trees where parsers can perform best with them.

We will evaluate the dependency trees obtained by using different versions of HPTs using a state-of-the-art parser (MaltParser[7]), which our parser implementation is largely based on[8]. Once we identify the best HPT version, then we will use the same trees for evaluating our parser in the following chapter.

5.1.3.1 The transformation of HPT outputs

It is important to note that the variation in parsing accuracy using any version of the HPT does not reflect the linguistic significance of ordering the items in the HPT. Although we prefer to use a version that helps to achieve the highest degree of parsing accuracy, it is fairly easy to change the organisation of the dependency trees produced by a version of HPT to a format that would be produced by a different HPT version. If we are using HPT V.3, which gives coordinated conjunction a higher priority than nouns, the result will be identical to the tree in Figure 5.6(1). If we are using HPT V.4, which gives coordinated conjunction a lower priority than nouns, then the result will be the same as the tree in Figure 5.6(2).

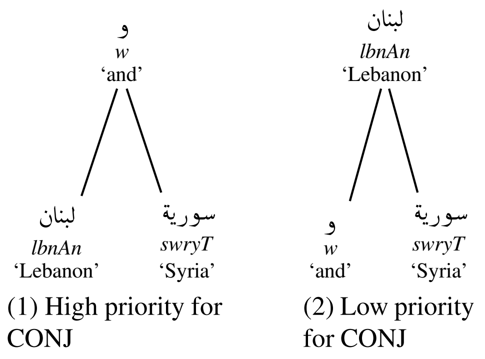


Figure 5.6: Different treatment of sentence-initial conjuction.

Here we will show an approach for transforming differently organised dependency

[7]We have tested MaltParser with the default settings of the Nivre arc-standard algorithm [Nivre, 2008] because we are implementing our parser based on this algorithm.

[8]The modifications made to the arc-standard algorithm of MaltParser are explained in Section 5.2.

trees that are produced by different versions of HPT, moving from one representation to another by using a simple algorithm that uses a rewrite rule. We demonstrate the way we can reorganise tree structures produced by HPT V.3 (which gives 'CONJ' and 'ICONJ' the highest priority) to tree structures that would have been produced by HPT V.4 (which gives 'CONJ' and 'ICONJ' the lowest priority). For example, the rewrite rule in (12) can be used for transforming the tree in Figure 5.7(1) to the tree in Figure 5.7(2). A tree is traversed and checked if there is a rewrite rule that matches it, and the rewrite rule is then applied to it. This subsequently reorganises its structure accordingly. The structure of the tree in Figure 5.7(1) matches the left-hand-side of a rule in 12, and thus we can reorder the tree element to match the other side of the rewrite rule. The same process but with different rewrite rules can be used for transforming trees produced by HPT V.5 to trees with determiners heading nouns and other variations induced by changes to the HPT.

(12) $CONJ(X(X_D), Y(Y_D)) \rightarrow X(X_D, CONJ, Y(Y_D))$

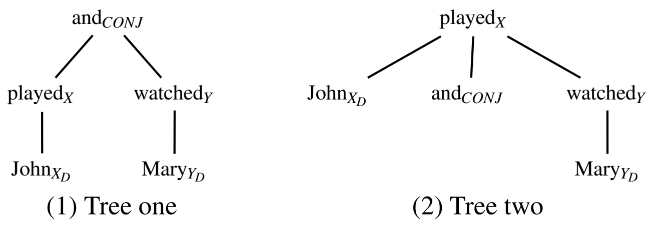


Figure 5.7: Different forms of dependency tree organisation for the sentence `John played and Mary watched`.

5.1.3.2 The production of different HPT versions

From our previous observations of converting PATB trees to dependency structures in Section 5.1.2 we will produce the following versions of HPT (see Appendix D for the organisation of heads in each HPT version):

1. HPT V.1: The ordering of the elements inside the HPT is performed randomly during the conversion procedure.

2. HPT V.2: This is the same as HPT V.1 but splits coordinate conjunctions (CONJ) into CONJ and ICONJ where ICONJ is used for sentence-initial coordinates and CONJ is used for non-sentence-initial coordinates.

3. HPT V.3: This is the same as HPT V.2 but items in the entries are ordered according to their phrase category. For example, adjectives in adjectival phrases are put before nouns or prepositions. Additionally, coordinating conjunctions (CONJ and ICONJ) have the highest priority inside all the entries and determiners are given higher priority than nouns. It could be argued that coordinating conjunctions should be treated either as heads or as modifiers in conjunction phrases. Moreover, we can see that coordinated conjunctions are treated differently in PADT and CATiB treebanks, as shown in Figures 5.8(2) and 5.8(3) respectively for the sentence in (13). In PADT the coordinating conjunction is treated as the head of the conjunction, whereas in CATiB the first conjunct is treated as the head of the coordinated conjunction where the coordinated conjunction heads the second conjunct. If we give coordinated conjunctions a high priority then we get the tree in Figure 5.8(4), where the coordinated conjunction is the head of the first conjunct in the conjunction.

(13) An Arabic sentence ([Habash and Roth, 2009c])[9]

خمسون الف سائح زاروا **لبنان و سورية** في أيلول الماضي

"xmswn 'lf sA'i.h zArwA **lbnAn w swry'** fy 'ylwl almaDy"

Fifty thousand tourists visited **Lebanon and Syria** last September

4. HPT V.4: Same as HPT V.3 but coordinating conjunctions (CONJ and ICONJ) have the lowest possible priority inside all the entries. If we give CONJ and ICONJ a low priority in the HPT then we get a tree like that in Figure 5.8(5), where the noun heads the conjunction phrase.

5. HPT V.5: Some theories, such as categorial grammar, treat the determiner as the head of the noun in NPs while others, such as GPSG [Gazdar et al., 1985] and HPSG [Pollard and Sag, 1994], treat the noun as the head of determiners. In Arabic, there are no indefinite articles to be treated as determiners and the definite articles, which are determiners, are attached to nouns. Thus most of the determiners (definite articles) in PATB are treated as a part of the noun. For example, ال *the* in the word الباص *AlbA.s* 'the bus' is a definite article attached to the noun 'bus'. However, there are some numbers and demonstrative determiners in the

[9]We use this example to show the difference between our dependency tree and those in PADT and CATiB.

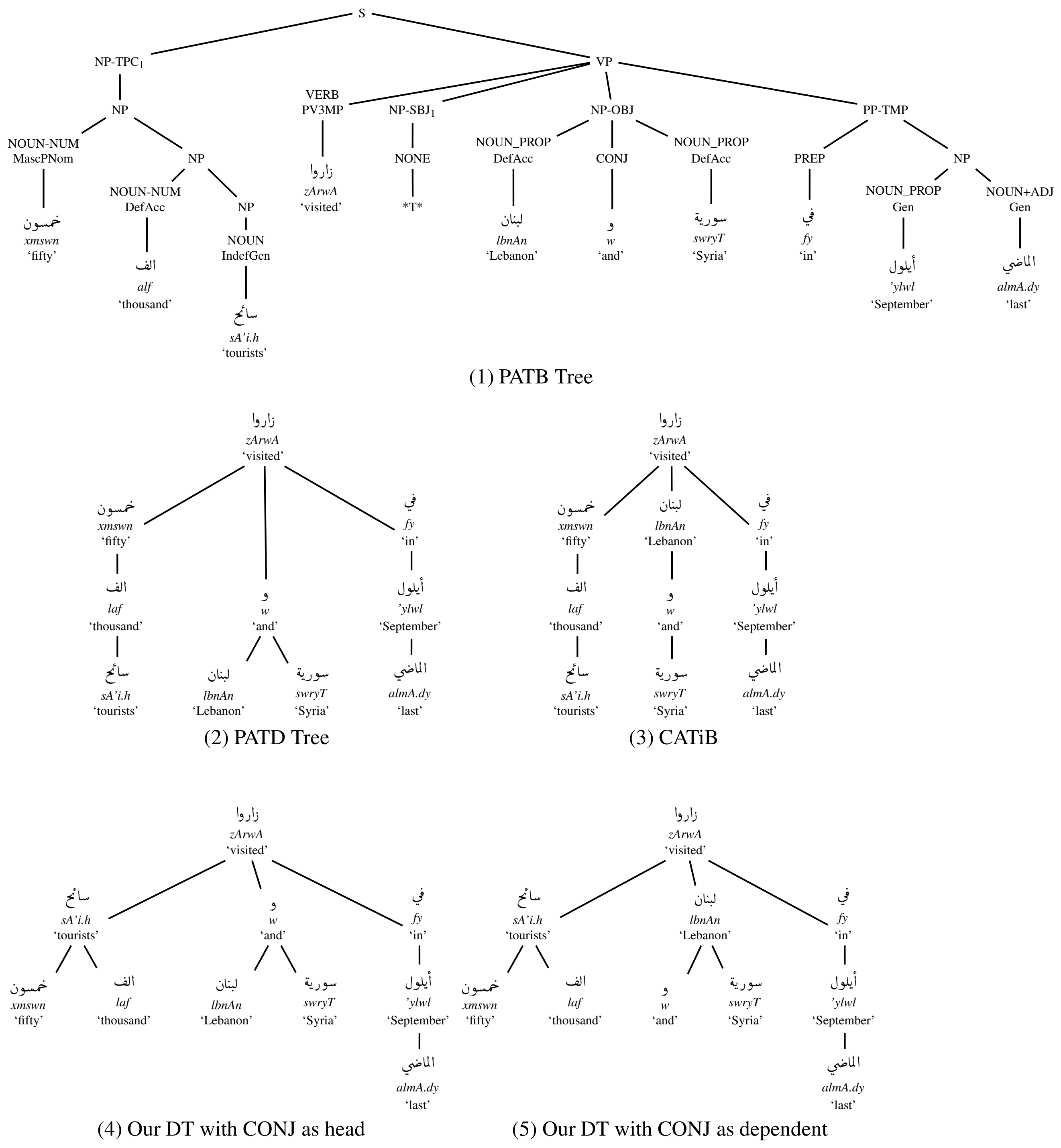


Figure 5.8: A comparison of a PATB tree with trees in PADT, CATiB, and ours (without POS tags and functional labels) for the sentence in example (13).

PATB and giving them priorities to be the heads of nouns could affect parsing accuracy. In this version we give nouns a higher priority than determiners and everything else is the same as HPT V.3.

5.1.3.3 5-fold cross validation

We split the dependency treebank, which we have obtained by converting PATB (for convenience, we will call the converted PATB 'DATB' from now on), into training data and testing data.

In order to perform a 5-fold cross validation, we can systematically generate five sets of testing data and five sets of training data from the dependency treebank. For example, the first set of testing data contains the first 20% of the DATB sentences and the remaining 80% of the sentences are used as training data, the second set of testing data consists of the second 20% of the DATB sentences and the remaining 80% of the sentences are used for training data, and so on. The training data for each fold contains about 112,800 tokens while the testing data for each fold contains about 28,000 tokens. The average length of the sentences is 29 words. The total number of testing sentences in each fold is about 970 and the total number of training sentences in each fold is about 3870.

We will use the dependency trees generated by using different HPT versions for evaluating the arc-standard algorithm of MaltParser because this is one of the state-of-the-art parsers and our parser is largely based on this algorithm. We compare the output of MaltParser against a gold-standard POS tagged data set, which we obtain from DATB, to measure the accuracy of the parser. The term *accuracy* is used as a measure for indicating the correctness of parser output by comparing it against a gold-standard data. We present three accuracy measures: (i) **labelled attachment score (LAS)**, which is the percentage of the correct dependency relations with the correct labels of the dependency relations (DEPREL) between tokens as shown in Table C.1 in Appendix C, (ii) **unlabelled attachment score (UAS)**, which is the percentage of the correct dependency relation (i.e., the percentage of tokens with the correct head) regardless of the DEPREL, and (iii) **label accuracy (LA)**, which is the percentage of tokens with the correct dependency label.

We obtain the accuracy scores by using the scoring of the CoNLL evaluation software from the CoNLL-X shared task, *eval.pl*[10], with the default settings which exclude the punctuation tokens in computing the scores for LAS, UAS, and LA. Additionally,

[10]Available at: http://ilk.uvt.nl/conll/software.html#eval.

we evaluate the parsing **efficiency** by measuring the parsing time in seconds per dependency relations between two tokens.

We obtain the LAS, UAS, and LA results for all the folds and then we divide the total for each of the LAS, UAS, and LA by the number of folds to get the average accuracy rate of each one of them.

The results for testing MaltParser on different versions of the HPT are shown in Table 5.3. We have conducted all of the testing on the MaltParser at this stage because we would like to find out which version of the HPT leads to the best parse results. Then, we will use the DATB produced by that version of the HPT for performing a number of evaluations on our parser in Chapter 6, which we will compare against the results we have obtained for MaltParser. From Table 5.3 we can identify that the parser performs best when using HPT V.5 for converting PATB data to the dependency format. This version of HPT is presented in Figure 5.9

HPT	MaltParser			Speed
	UAS (%)	LAS (%)	LA (%)	Second/Relation
V.1	56.7	52.5	86.96	0.171
V.2	57.8	53.1	87.12	0.2
V.3	74.9	69.8	92.17	0.146
V.4	74.8	69.6	91.98	0.163
V.5	**75.2**	**70.0**	**92.2**	**0.144**

Table 5.3: MaltParser performance when trained and tested on different versions of HPT.

- V.1 Random head-child ordering.
- V.2 Same as (V.1) but sentence-initial CONJ is marked as ICONJ.
- V.3 Same as (V.2) but items are ordered based on their phrase category and coordinated items are given the highest priority in every HPT entry, and determiners are given higher priority than nouns.
- V.4 Same as (V.3) but coordinated items are given the lowest priority in every entry.
- V.5 Same as (V.3) but nouns have higher priority than determiners. This version leads to more accurate parsing and is much faster than the other versions.

POS tag	Possible head-child(s)
ADJP	CONJ, CONJP, ADJ, DET+ADJ, ADJP, NOUN, DET+NOUN, DET, NP
ADJP-OBJ	ADJ
ADJP-PRD	CONJ, ADJ, ADJP, NOUN, NP
ADV	CONJ, ADV, VP, PV, VERB, S, NOUN, DET+NOUN, NP, NP-SBJ, REL_PRON, PRON, NOUN_NUM, PART, ADJ, DET+ADJ, PP, NUM, PART, SUB_CONJ, SBAR
CONJP	ICONJ, CONJ, NOUN, ADJ, PART, PREP
FRAG	CONJ, SBAR, NOUN, NP, PRT, ADV, PP, FRAG
NAC	CONJ, NOUN, NP, ADV, SUB_CONJ
NP	ICONJ, CONJ, NOUN, NOUN_PROP, DET+NOUN, DET, NP, POSS_PRON, DEM_PRON, PRON, PV, SBAR, PART, PRN, ADV, ADJ, DET+ADJ, PREP, PP, QP, NOUN_NUM, NUM, ABBREV, VP, S, FOREIGN
NP-OBJ	CONJ, NOUN, NOUN_PROP, DET+NOUN, DET, NP, PRON, VP, S, ADJ, DET+ADJ, QP, NOUN_NUM, NUM, ABBREV
NP-PRD	CONJ, NOUN, NOUN_PROP, NP, PRON, PART, ADJ, QP, NOUN_NUM, NUM, ABBREV
NP-SBJ	CONJ, NOUN, NOUN_PROP, DET+NOUN, DET, NP, PRON, SBAR, PRT, PV, ADJ, DET+ADJ, QP, ABBREV
NP-TPC	CONJ, NOUN, NOUN_PROP, NP, PRON, ADV, ADJ, NUM, PUNC
PP	ICONJ, CONJ, PREP, PP, NOUN, DET+NOUN, NOUN_PROP, SBAR, PV, VP, X, NP, S, NEG_PART
PP-OBJ	PREP, PP, NOUN
PP-PRD	CONJ, PREP, PP, NOUN, NP, S, PART, SBAR
PP-SBJ	PREP
PRN	CONJ, S, NP, ADV, PP, NOUN, NOUN_PROP, DET+NOUN, NUM, SBAR, ADJ, ADJP
PRT	ICONJ, VERB, PART, PRT
QP	CONJ, NOUN, NOUN_NUM, NUM, ADJ
S	ICONJ, S, VERB, PV, IV, VP, CONJP, CONJ, NP, NP-TPC, NP-SBJ, NP-PRD, PRT, SBAR, ADV, ADJP, ADJP-PRD, PP, PP-PRD, PART, FRAG, FRAG-PRD, X
S-PRD	VP
S-SBJ	VP
SBAR	ICONJ, CONJ, VERB, PV, INTERJ, NOUN, S, PART, PRON, SBAR, UCP, SUB_CONJ, WHNP
SBAR-SBJ	SBAR, S, SUB_CONJ
SBARQ	SQ, S
SBARQ-PRD	S
SQ	NP-SBJ
UCP	CONJ, S, NP, SBAR, VP, ADV, ADJP
UCP-OBJ	CONJ, NP
UCP-PRD	CONJ, NP
UCP-SBJ	CONJ, NP
VP	ICONJ, CONJ, PRT, VERB, PV, IV, CV, VP, PART, X, S, SBAR, NOUN, NOUN_PROP, NP, NP-SBJ, NP-PRD, NP-OBJ, NP-TPC, ADJ , ADV, PREP, PP, NEG_PART, NAC
WHNP	CONJ, NOUN, NP, PRON, PREP, PP, WHNP
X	ICONJ, CONJ, PV, IV, PART, NOUN, NP, NOUN_PROP, PRON, PREP, NEG_PART, NUM, ABBREV, NO_FUNC, PUNC

Figure 5.9: HPT V.5, similar to V.3 but nouns have higher priority than determiners.

5.1.3.4 A brief error analysis

The different treatments of CONJ and NP items in the HPT affect the performance of MaltParser. We analyse the errors made by MaltParser by looking at the number of words and the types of words for which the parser failed to assign the correct head or the correct dependent. The CoNLL evaluation output contains precisely the number of incorrectly assigned heads, dependents, and head and dependent for specific words. In Table 5.3 we can observe that when marking the sentence-initial conjunctions as 'ICONJ', the parsing accuracy improves by 0.54% for LAS, 1.17% for UAS, and 0.16% for LA. There may be a number of reasons for this improvement in accuracy. Firstly, it is likely that the parser does not assign incorrect items to sentence-initial conjunctions when they are marked as ICONJ. Thus the CoNLL evaluation software did not include the ICONJ in the errors in Figure 5.10 since not many items are assigned to ICONJ incorrectly, which confirms this analysis. We may also notice that the number of errors regarding 'CONJ' is significantly reduced. The parser assigned incorrect dependents, and incorrect head and dependent to `w/CONJ` twice only compared with twenty one times when the sentence-initial `CONJ` was not marked as `ICONJ`. Other items that the parser made slightly less mistakes with are `fy/PREP` and `b/PREP`.

Moreover, an interesting effect on the parsing accuracy when marking sentence-initial conjunctions as 'ICONJ' is that the parser may assign incorrect head or dependent to different types of words. For example, as can be seen from Figure 5.10, the parser assigns incorrect heads or dependents to `An/SUB_CONJ` instead of `mn/PREP` when sentence-initial CONJ is marked as ICONJ. It is also worth noting that the number of incorrect heads, dependents, or both assigned to `An/SUB_CONJ` is less than those assigned to `mn/PREP`, which could be another factor in terms of improved accuracy.

A persistent pattern when sentence-initial conjunction is marked as 'ICONJ' is that the assignments of incorrect dependents to heads are reduced for all words in Figure 5.10(2), while other relations (`head`, `any`, and `both`) have also been reduced for most of the words.

In the following sections, we will describe our implementation of MaltPaser and will give a detailed description of how to apply various constraints to improve its performance.

```
        | any | head | dep | both
--------+-----+------+-----+-------------
w/CONJ  | 864 | 864  | 21  | 21
fy/PREP | 837 | 674  | 554 | 391
b/PREP  | 397 | 391  | 54  | 48
l/PREP  | 396 | 379  | 65  | 48
mn/PREP | 327 | 295  | 114 | 82
```

(1) Top 5 word errors when MaltParser used HPT V.1

```
            | any | head | dep | both
------------+-----+------+-----+-----------
w/CONJ      | 851 | 851  | 2   | 2
fy/PREP     | 825 | 672  | 534 | 381
b/PREP      | 407 | 402  | 52  | 47
l/PREP      | 390 | 376  | 64  | 50
An/SUB_CONJ | 331 | 330  | 11  | 10
```

(2) Top 5 word errors when MaltParser used HPT V.2

Figure 5.10: Parse errors when using different HPT versions.

5.2 A basic shift-reduce parser

We base our parser implementation on the arc-standard algorithm of MaltParser. The arc-standard algorithm is the closest correspondent to the shift-reduce parsing algorithm, which we have explained in Section 2.5. There are three parse operations in the arc-standard algorithm, LEFT-ARC, RIGHT-ARC, and SHIFT. These operations are applied to a queue of words which have not yet been looked at and a stack of words which have been inspected but have not yet been assigned a syntactic role.

The LEFT-ARC and the RIGHT-ARC operations correspond to the reduce action of the shift-reduce algorithm. These two operations establish head-dependent relations between the head item of the queue and the top item on the stack. The SHIFT operation is exactly the same as the shift action in the shift-reduce algorithm. The LEFT-ARC and the RIGHT-ARC operations are applied to one node in a queue of input strings and one node on the stack. The LEFT-ARC operation makes the first node in the queue the parent of the top node on the stack and pop the stack while the RIGHT-ARC operation makes the top node on the stack the parent of the first node in the queue, pop the queue and returns the item on the top of the stack to the queue.

The original implementation of the arc-standard algorithm uses a deterministic approach for parsing natural language text, where a support vector machine (SVM) classifier [Chang and Lin, 2011] is used for learning parse operations from a dependency treebank. The classifier helps the parser to predict the most likely correct parse operation when it is presented with a non-deterministic choice between multiple parse operations. As Nivre [Nivre, 2008] states, in this kind of implementation the parser derives a single parse analysis by incrementally selecting a parse operation, which makes the parsing process very simple and efficient. Moreover, by using an appropriate classifier, a good parsing accuracy is achievable [Nivre, 2008, p. 514] and he shows that the arc-standard algorithm produces better accuracy than other algorithms of MaltParser (such as the arc-eager algorithm) for Arabic, where for unlabelled attachment score with arc-standard the parser achieves 77.76% while with arc-eager it achieves

76.31%, and for labelled attachment score it achieves 65.79% with arc-standard and 64.93% with arc-eager.

The main difference between the arc-standard algorithm and the arc-eager algorithm is that the former has an extra parse operation (the REDUCE operation) in addition to the LEFT-ARC, RIGHT-ARC, and SHIFT operations. In the arc-eager algorithm, the RIGHT-ARC operation attaches the right dependents to a head before the right dependents find all of their own right dependents. The parent and the dependent are stored on the stack for further processing by the RIGHT-ARC operation and they are removed from the stack by the REDUCE operation at a later time.

The original arc-standard algorithm uses a deterministic approach to parsing natural language texts. The parser follows suggestions made by a parse model to perform a specific parse action (LEFT-ARC, RIGHT-ARC, or SHIFT) at each parse step. Performing the wrong parse action at a particular step during parsing will have a knock on effect on subsequent parsing steps. Hence, the error propagation could be substantial. Using a non-deterministic approach, where the parser is presented with multiple actions to take, allows the parser to recover from a previous mistake if this is subsequently identified.

We re-implement the arc-standard algorithm in a way that lets us run it both deterministically and non-deterministically. We will call our parser **DNDParser**, which is short for deterministic and non-deterministic data-driven parser. At each parse step, we generate a state for LEFT-ARC, RIGHT-ARC, and SHIFT, and we will assign different scores to each state. Each score is computed according to the recommendations made by the parsing model. For example, if the parse model recommends a SHIFT action when the parser is generating a SHIFT state then we can give the state a score of a positive number. Otherwise we give a score of 0, and the same computation is used for scoring LEFT-ARC and RIGHT-ARC states.

5.2.1 Non-projective parsing

With the original implementation of the arc-standard algorithm it is not possible to produce non-projective parse trees. Using the parse transitions in Figure 5.12, we show that using the standard LEFT-ARC, RIGHT-ARC and SHIFT operations it is not possible to parse the non-projective Czech sentence in Figure 5.11.

In Figure 5.12 we demonstrate the transitions of the original algorithm for parsing the non-projective Czech sentence in 5.11 in an attempt to reproduce the dependency graph in Figure 5.11 using the dependency relations that exist between the words in the sentence. These dependency relations, which we extract from the dependency graph in Figure 5.11, are displayed above the parse transitions in Figure 5.12[11].

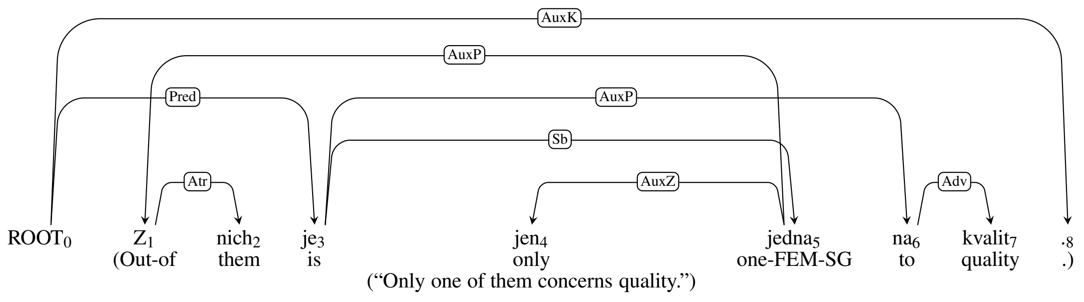


Figure 5.11: Non-projective dependency graph for a Czech sentence from the Prague Dependency Treebank. [Nivre et al., 2010].

```
Dependency relations: (0 > 3)  (0 > 8) (1 > 2) (3 > 5) (3 > 6) (5 > 1) (5 > 4) (6 > 7)
-------------------------------------------------------------------------------------------------
Steps   Action           Queue                      Stack               Arcs
-------------------------------------------------------------------------------------------------
1       θ              [0,1,2,3,4,5,6,7,8]          []                  θ
2       SHIFT          [1,2,3,4,5,6,7,8]            [0]                 θ
3       SHIFT          [2,3,4,5,6,7,8]              [1,0]               θ
4       RIGHT-ARC      [1,3,4,5,6,7,8]              [0]                 A1=(1 > 2)
5       SHIFT          [3,4,5,6,7,8]                [1,0]               A1
6       SHIFT          [4,5,6,7,8]                  [3,1,0]             A1
7       LEFT-ARC       [5,6,7,8]                    [4,3,1,0]           A2=A1∪(5 > 4)
8        _             [5,6,7,8]                    [3,1,0]             A2

-------------------------------------------------------------------------------------------------
```

Figure 5.12: Parse transitions for processing the sentence in Figure 5.11 using the original arc-standard algorithm of MaltParser.

In step 8 as shown in Figure 5.12, the parser may perform LEFT-ARC, RIGHT-ARC, or SHIFT operations, but none of these operations leads to producing a tree matching the original non-projective tree. According to the set of dependency relations (as shown at the top of Figure 5.12), a LEFT-ARC operation is not allowed. If the parser performs a LEFT-ARC operation, it will lead to producing a tree that will not match the original tree because that will make 5 the head of 3, which does not match any relations in the original tree. If the parser performs RIGHT-ARC operation is allowed which will make 3 the head of 5 but at this stage this is not an ideal operation because 5 will not be available in subsequent stages when it is required to become the

[11] Although we show that the parser fails in Figure 5.12, the algorithm will not fail in reality because it will assign the head of the queue as the parent or as the daughter of an item on the top of the stack and it will produce a graph. However, the graph will not be the actual graph that we attempt to reproduce.

head of `1`, which remains on the queue[12]. This means that `1` will subsequently receive the wrong head, which leads to producing a tree that does not match the original tree. SHIFT operation will produce a state where `5` is at the top of the stack, which means that both `5` and `3` are placed on the stack. This means that they will never be in a state where `3` can be assigned as the head of `5` and thus the parser will not produce a tree that matches the original tree.

We extend the original algorithm to allow for this limitation. Our extension allows the LEFT-ARC and the RIGHT-ARC operations to combine the head of the queue with an item that may or may not be at the top of the stack, and thus we have more complex ARC operations, which are LEFT-ARC(*N*) and RIGHT-ARC(*N*) where *N* is the position of the item on the stack which is combined with the head of the queue. Moreover, post LEFT-ARC(*N*) or RIGHT-ARC(*N*) operations, the extended algorithm will move *N* items back to the queue from the stack. For example, if the head of the queue is combined with the third item on the stack (i.e., after LEFT-ARC(`3`)), then the top two items on the stack are moved back into the queue.

```
Dependency relations: (0 > 3)  (0 > 8) (1 > 2) (3 > 5) (3 > 6) (5 > 1) (5 > 4) (6 > 7)
-------------------------------------------------------------------------------------------
Steps   Action              Queue                         Stack           Arcs
-------------------------------------------------------------------------------------------
1       θ                   [0,1,2,3,4,5,6,7,8]           []              θ
2       SHIFT               [1,2,3,4,5,6,7,8]             [0]             θ
3       SHIFT               [2,3,4,5,6,7,8]               [1,0]           θ
4       RIGHT-ARC(0)        [1,3,4,5,6,7,8]               [0]             A1=(1 > 2)
5       SHIFT               [3,4,5,6,7,8]                 [1,0]           A1
6       SHIFT               [4,5,6,7,8]                   [3,1,0]         A1
7       SHIFT               [5,6,7,8]                     [4,3,1,0]       A1
8       LEFT-ARC(0)         [5,6,7,8]                     [3,1,0]         A2=A1∪(5 > 4)
9       LEFT-ARC(1)         [5,6,7,8]                     [3,0]           A3=A2∪(5 > 1)
10      RIGHT-ARC(0)        [3,6,7,8]                     [0]             A4=A3∪(3 > 5)
11      SHIFT               [6,7,8]                       [3,0]           A4
12      SHIFT               [7,8]                         [6,3,0]         A4
13      RIGHT-ARC(0)        [6,8]                         [3,0]           A5=A4∪(6 > 7)
14      RIGHT-ARC(0)        [3,8]                         [0]             A6=A5∪(3 > 6)
15      RIGHT-ARC(0)        [0,8]                         []              A7=A6∪(0 > 3)
16      SHIFT               [8]                           [0]             A7
17      RIGHT-ARC(0)        [0]                           []              A8=A7∪(0 > 8)
18      SHIFT               []                            [0]             A8
19      θ                   []                            [0]             A8
-------------------------------------------------------------------------------------------
```

Figure 5.13: Parse transitions for processing the sentence in Figure 5.11 using a modified arc-standard algorithm of MaltParser.

By extending the algorithm to allow for combining the head of the queue with an appropriate item that may be buried in the stack, we can reproduce the non-projective

[12]The LEFT-ARC and the RIGHT-ARC operation will remove the dependent item from the queue or the stack and hence the dependent item will not be available in subsequent parsing steps.

graph shown in Figure 5.11, given the set of dependency relations extracted from the graph. This is our contribution C.4 of this study. The parse transitions of the extended algorithm, as shown in Figure 5.13, reproduce the non-projective graph shown in Figure 5.11. The line in bold in step `9` of Figure 5.13 shows the parse transitions that the original algorithm would not have performed. In step `8`, the extended algorithm combines the head of the queue with the second item on the stack using the LEFT-ARC(*1*) operation[13].

Our approach contrasts with the treatment of non-projective dependencies in the standard non-projective MaltParser algorithm, which is a list-based algorithm.

The list-based algorithm uses two lists, a queue of input strings, and a set of dependency arcs. The queue stores the tokens from the input buffer β, the two lists (λ_1 and λ_2) store partially processed tokens; i.e., tokens that have been removed from β where they may be potential daughters of the head of the buffer β, and the set of dependency arcs store the dependency links between two tokens (heads and dependents).

There are four parse operations in the list-based algorithm:

- **LEFT-ARC** produces an arc $A(j, l, i)$, and adds it to graph G, where j is the first node in the β, i is the head of the list λ_1, and l is the dependency label. This assigns j as the head of i and it moves i from the list λ_1 to the list λ_2 if three conditions hold: (i) i is not the artificial root node, (ii) i does not already have a parent, and (iii) a path from i to j does not exist in the graph G where G is a set of Arcs;
- **RIGHT-ARC** produces an arc $A(i,l,j)$ by assigning i as the head of j, and adds it to graph G, where l is the dependency label. j is first node in the β and i is the head of the list λ_1, which is moved from the list λ_1 to the list λ_2 if two conditions hold: (i) j does not already have a head, and (ii) there does not exist a path from j to i in the graph G;
- **NO-ARC** in this operation, the first item of the list λ_1 is popped and pushed at the start of the list λ_2.
- **SHIFT** This operation performs three tasks: (i) it generates a new list λ_1 by concatenating list λ_1 and list λ_2 , (ii) it empties list λ_2, and (iii) it removes the first node from the β and puts it on the head of the newly generated list λ_1.

[13]The head of the queue was not combined with the top item on the stack in step `9` because that would have removed `5` from the queue which is needed later to be used as the head of `1`.

The algorithm considers every pair of nodes for generating arcs whenever possible by storing the parent and the daughter of a dependency relation in either list λ_2 or queue β. The NO-ARC operation, which moves a node from list λ_1 to list λ_2 even if the node does not yet have a parent, allows for the production of non-projective parse tree but the conditions for generating an arc are: (i) the daughter is not the artificial root node, (ii) the daughter does not already have a head, and (iii) there is no path connecting the daughter to the parent. In Figure 5.14 we reproduce the non-projective graph shown in Figure 5.11 using this algorithm.

Dependency relations: (0 > 3) (0 > 8) (1 > 2) (3 > 5) (3 > 6) (5 > 1) (5 > 4) (6 > 7)

Steps	Action	List1	List2	Queue	Arcs
1	θ	[0]	[]	[0,1,2,3,4,5,6,7,8]	θ
2	SHIFT	[0,1]	[]	[2,3,4,5,6,7,8]	θ
3	RIGHT-ARC	[0]	[1]	[2,3,4,5,6,7,8]	A1=(1 > 2)
4	SHIFT	[0,1,2]	[]	[3,4,5,6,7,8]	A1
5	NO-ARC	[0,1]	[2]	[3,4,5,6,7,8]	A1
6	NO-ARC	[0]	[1,2]	[3,4,5,6,7,8]	A1
7	RIGHT-ARC	[]	[0,1,2]	[3,4,5,6,7,8]	A2=A1∪(0 > 3)
8	SHIFT	[0,1,2,3]	[]	[4,5,6,7,8]	A2
9	SHIFT	[0,1,2,3,4]	[]	[5,6,7,8]	A2
10	LEFT-ARC	[0,1,2,3]	[4]	[5,6,7,8]	A3=A2∪(5 > 4)
11	RIGHT-ARC	[0,1,2]	[3,4]	[5,6,7,8]	A4=A3∪(3 > 5)
12	NO-ARC	[0,1]	[2,3,4]	[5,6,7,8]	A4
13	LEFT-ARC	[0]	[1,2,3,4]	[5,6,7,8]	A5=A4∪(5 > 1)
14	SHIFT	[0,1,2,3,4,5]	[]	[6,7,8]	A5
15	NP-ARC	[0,1,2,3,4]	[5]	[6,7,8]	A5
16	NO-ARC	[0,1,2,3]	[4,5]	[6,7,8]	A5
17	RIGHT-ARC	[0,1,2]	[3,4,5]	[6,7,8]	A6=A5∪(3 > 6)
18	SHIFT	[0,1,2,3,4,5,6]	[]	[7,8]	A6
19	RIGHT-ARC	[0,1,2,3,4,5]	[6]	[7,8]	A7=A6∪(6 > 7)
20	SHIFT	[0,1,2,3,4,5,6,7]	[]	[8]	A7
21	**NO-ARC**	**[0,1,2,3,4,5,6]**	**[7]**	**[8]**	**A7**
22	**NO-ARC**	**[0,1,2,3,4,5,6]**	**[6,7]**	**[8]**	**A7**
23	**NO-ARC**	**[0,1,2,3,4,5]**	**[5,6,7]**	**[8]**	**A7**
24	**NO-ARC**	**[0,1,2,3]**	**[4,5,6,7]**	**[8]**	**A7**
25	**No-ARC**	**[0,1,2]**	**[3,4,5,6,7]**	**[8]**	**A7**
26	**No-ARC**	**[0,1]**	**[2,3,4,5,6,7]**	**[8]**	**A7**
27	**No-ARC**	**[0]**	**[1,2,3,4,5,6,7]**	**[8]**	**A7**
28	RIGHT-ARC	[]	[0,1,2,3,4,5,6,7]	[8]	A8=A7∪(0 > 8)
29	SHIFT	[0,1,2,3,4,5,6,7,8]	[]	[]	A8
30	θ	[0,1,2,3,4,5,6,7,8]	[]	[]	A8

Figure 5.14: Parse transitions for processing the sentence in Figure 5.11 using the list-based non-projective MaltParser algorithm.

Our extended algorithm is different from the list-based algorithm in a number of ways: (i) the list-based algorithm uses three data-structures (queue, list one λ_1 and list two λ_2) whereas our algorithm uses two data structures (queue and stack), (ii) the list-based algorithm uses an extra operation, which is NO-ARC operation, that operates on both λ_1 and λ_2, (iii) the SHIFT operation in the list-based algorithm performs more complex operations than the SHIFT operation in our algorithm. It concatenates the data stored in λ_1 and λ_2 and produces a new λ_1; it empties λ_2; and then it moves the head of the queue to the front of the newly generated λ_1.

The operations in our algorithm are computationally less expensive than the operations in the list-based algorithm. Our SHIFT operation is much simpler than the one in the list-based algorithm. The SHIFT operation in the list-based algorithm involves three actions: concatenating λ_1 and λ_2, emptying λ_2, then moves the head of the queue to the front of the newly generated λ_1. The SHIFT operation in our algorithm involves one action only, moving the head of the queue onto the top of the stack. Additionally, the list-based algorithm uses one extra arc operation (NO-ARC) which moves the head of λ_1 to the front of λ_2. This operation is performed until it is possible to do either a LEFT-ARC or a RIGHT-ARC, or until there are no items left on λ_1. As we can see from Figure 5.14, in the steps from 21 to step 27 marked in bold, there are seven NO-ARC transitions before the RIGHT-ARC operation is performed. In our algorithm, there is no requirement for performing this extra arc operation since the LEFT-ARC and the RIGHT-ARC operations perform one left-to-right pass over the stack operation, which results in fewer parse transitions than the number of parse transitions performed by the list-based algorithm for parsing the non-projective sentence in Figure 5.11.

Other non-projective dependency parsing techniques proposed by Kuhlmann and Nivre [Kuhlmann and Nivre, 2010] include the following:

- Pseudo-projective parsing: The approach of converting a projective dependency parser to a non-projective parser involves two main steps: (i) a pre-processing step, which involves transforming the original non-projective trees into projective trees. In this step each non-projective arc is replaced with a projective arc. For example, a non-projective arc (i,l,j) is replaced with a projective arc (k,l,j) where the closest transitive head of i is the head k. The training data is encoded with rich information to allow the parser to undo these changes, so that the projective parser is trained on it as usual. (ii) a post-processing step, where the encoded information in the training data, is used for deprojectivising the projective trees produced by the projective parser to non-projective trees.

- Non-adjacent arc transitions: In this approach the parser is allowed to create arcs between non-neighbouring arcs. This is achieved by extending the arc-standard to do the LEFT-ARC-2_l and the RIGHT-ARC-2_l operations, which are:

LEFT-ARC-2$_l$: This operation creates an arc by making the top item on the stack the head of the third topmost item on the stack, and removes the third topmost item from the stack.

RIGHT-ARC-2$_l$: This operation creates an arc by making the third topmost item on the stack the head of the topmost item on the stack, and removes the topmost item from the stack.

Although Attardi [Attardi, 2006] claims that LEFT-ARC-2$_l$ and RIGHT-ARC-2$_l$ are sufficient for producing every non-projective tree Kuhlmann and Nivre [Kuhlmann and Nivre, 2010, p. 6] argues to the contrary.

- Online reordering: This approach is similar to the projective arc-standard parsing, where arcs are only added between two neighbouring items using a projective parser, but the difference is that the parser is allowed to reorder the items during parsing. The projective parser is extended with a SWAP operation, which basically reorders the position of the two topmost items on the stack. During the SWAP operation, the second topmost item from the stack is returned to the queue.

5.2.2 Assigning scores to parse states

Our parsing algorithm generates different parse states using SHIFT, LEFT-ARC, and RIGHT-ARC operations for each given parse state. A score is computed for each newly generated parse state by computing two different scores: (i) a score that is based on the recommendation made by a parse model. For example, for a SHIFT state the parser gives a score of 1 if a SHIFT operation is recommended by the model. Otherwise a score of 0 is given (and the same applies to LEFT-ARC and RIGHT-ARC operations). (ii) the score of the given state (which is the state that the new parse state is generated from). The sum of these two scores is assigned to the newly generated parser state.

Using only recommendations from a parse model to assign scores to newly generated states makes the parser operate exactly as the standard deterministic algorithm because the state with the highest score is obtained by following the parsing model.

The advantage of assigning a score to each parse state is that: (i) we can manipulate the assignment of various other scores for applying different constraint rules to parse states. This way we can apply different constraint rules to the data-driven parser, which correspond to our contribution C.1, and (ii) we can rank a collection of parse states by using their scores and then process the state with the highest score, which we consider to be the most plausible state.

5.2.3 Parsing with dynamic programming and chart tables

Since the reimplementation of the arc-standard algorithm may produce a large set of parse states from a given state, whereby each state generates a parse analysis, we will have to efficiently process a potentially large set of states. The most popular algorithm that can efficiently process a potentially large set of parse states is dynamic programming and a chart table. This has been used in MSTParser (see Section 3.1.2.1 for a description of MSTParser).

We use a treebank-induced classifier as a statistical model for recommending a parse action at each parse step, which results in generating a new parse state. For each parse state we assign a score (see Section 5.2.2). Dynamic programming is used for ranking competing states with respect to their plausibility (the plausibility of a state is based on its score.) The ranked states are then stored in an agenda[14] and the most plausible state (i.e., the state with the highest score) is explored by the parser. The scores that are given to each state are determined by suggestions made by a parse model. If a parse model suggests an action then a positive score is given to the state otherwise a score of *0* is given to the state, and the states are then ranked in the agenda. The following section describes an approach for generating a parse model.

5.2.4 The generation of a parse model

As mentioned in the introductory section of this chapter, one of the fundamental elements of a data-driven parser is a 'parse model', which is used for guiding the parser when it processes new sentences. The most popular approach used in generating a parse model involves inducing a machine learning classifier on a treebank. The output of the classifier induction can then be used to generate a model that a parser can use as a guide for processing new sentences. The effectiveness of a parse model largely depends on the classifier's ability to correctly classify a given set of data. In order to find out the most effective machine learning classifier, we have experimented with a number of classification algorithms that are available in the 'WEKA' toolkit[15] for

[14]An agenda is a list containing parse states ranked by their scores, where a state with the highest score is placed on the top of the agenda.

[15]WEKA is a publicly available toolkit containing a large number of machine learning algorithms. It is available at: http://www.cs.waikato.ac.nz/ml/weka/downloading.html.

classifying a set of training data[16].

The following steps explain an approach for generating a parse model from a dependency treebank:

1. Collecting parse states: We use a collection of parsed trees to obtain a set of parse states, i.e., condition:action pairs where the condition is the state of the queue and stack (i.e., the items in the queue and the stack) and the action is the parse operation that the parser took (which is either SHIFT, LEFT-ARC, or RIGHT-ARC), by using the relations that make up the trees to specify an appropriate sequence of actions. Consider, for instance, the tree in Figure 5.15 for the sentence *'The cat sat on the mat'*.

```
       sat
      /   \
   cat     on
    |       |
   the     mat
            |
           the
```

Figure 5.15: Dependency tree for *'The cat sat on the mat'*.

Figure 5.16 shows the transitions that the parser uses for producing the tree for the sentence *'The cat sat on the mat'*. Note that whilst constructing the training data we will not perform any LEFT-ARC or RIGHT-ARC operations if a dependency daughter is the head of an item that is not inspected yet; as can be seen from step 6 of Figure 5.16 where we perform SHIFT instead of RIGHT-ARC, since performing RIGHT-ARC at this point would make it impossible to later make `on` the head of `mat` because RIGHT-ARC would remove `on` from the `Queue`, which must not be removed because it is the head of `mat` which is still in the `Queue`.

We can treat the sequences shown in Figure 5.16 as a set of data-points which indicate what the parser should do in a given situation – for instance, in a situation like in step 6 (marked in bold in Figure 5.16) the parser should use SHIFT instead of RIGHT-ARC for the reason explained above. Given a set of such data-points, it is possible to extract and record the parse states and train a classifier for building a parse model, which can be used for predicting what kind of action the

[16]The training data for the classifier is obtained by parsing 4000 sentences where we have used the dependency tree of each sentence as the grammar for parsing the target sentence. This way we have obtained the exact parse states the parser had for each sentence, where they are used as training data.

Dependency relations: (sat > cat) (sat > on) (cat > the) (on > mat) (mat > the)

Steps	Action	Queue	Stack	Arcs
1	θ	[the,cat,sat,on,the,mat]	[]	θ
2	SHIFT	[cat,sat,on,the,mat]	[the]	θ
3	LEFT-ARC	[cat,sat,on,the,mat]	[]	A1=(cat > the)
4	SHIFT	[sat,on,the,mat]	[cat]	A1
5	LEFT-ARC	[sat,on,the,mat]	[]	A2=A1∪(sat > cat)
6	**SHIFT**	**[on,the,mat]**	**[sat]**	**A2**
7	SHIFT	[the,mat]	[on,sat]	A2
8	SHIFT	[mat]	[the,on,sat]	A2
9	LEFT-ARC	[mat]	[on,sat]	A3=A2∪(mat > the)
10	RIGHT-ARC	[on]	[sat]	A4=A3∪(on > mat)
11	RIGHT-ARC	[sat]	[]	A5=A4∪(sat > on)
12	SHIFT	[]	[sat]	A5
13	θ	[]	[sat]	A5

Figure 5.16: Action sequence for analysing the sentence *'The cat sat on the mat'*.

parser can carry out when parsing new sentences; i.e., it can be used for guiding the parser.

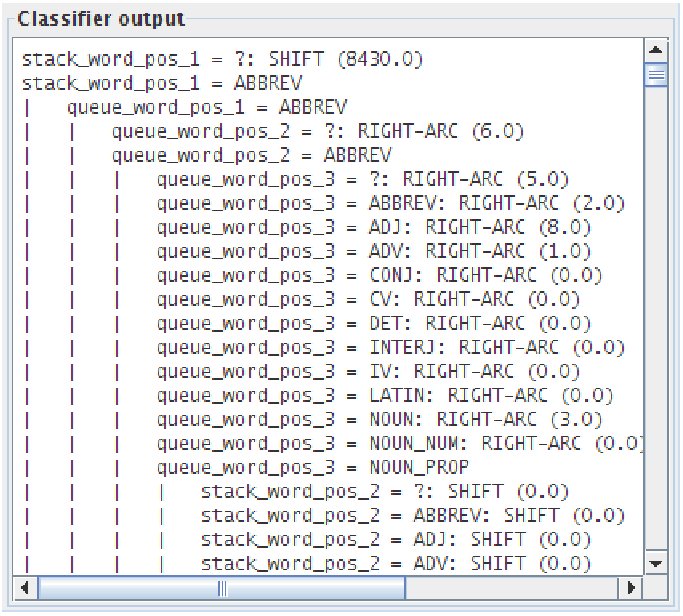
2. Preparing recorded parse states for classification: From the set of parse states that we obtain in step 1, we populate an *.arff* file with the correct data format, i.e., the format that is accepted by WEKA. An example of a set of WEKA-style data format is shown in Figure 5.17, which is based on the parse states shown in Figure 5.16. Here we have extracted the word forms as a feature for learning but it is possible to use a number of different features (such as POS tags, word position etc.) as values for the queue and the stack attribute parameters. Additionally, one can use different window sizes for the queue and the stack in the data selection as instances for the classification algorithms to learn from them. In Table 5.17, we use the window sizes of two items for the queue and two items for the stack, while the dash mark (`'-'`) represents an empty item where the queue or the stack did not contain an item in the given position.

3. Training a classifier using the *.arff* file: We supply WEKA with the data prepared in step 2 (i.e., the *.arff* file) and then we select a classification algorithm for learning. In Section 6.1 we have experimented with a large number of classification algorithms, where the J48 classifier produced the highest classification accuracy. Figure 5.18 is a screenshot of the J48 classification algorithm output from WEKA. The classification rules inferred by a decision tree classifier take the form of questions, such as *Is the POS tag on the top of the stack (ABBREV)?*, and a possible answer where the possible answer is a further question such as (*Is POS tag in the head of queue (ABBREV)?*) or a classification such as (*This is the

```
@relation states
@attribute queue_word_pos_1{‘the’,‘cat’,‘sat’,‘on’,‘mat’,‘-’}
@attribute queue_word_pos_2{‘cat’,‘sat’,‘on’,‘the’,‘mat’, ‘-’}
@attribute stack_word_pos_1{‘-’, ‘the’,‘cat’,‘sat’,‘on’}
@attribute stack_word_pos_2{‘-’,‘sat’,‘on’}
@attribute parse_action{‘SHIFT’, ‘LEFT-ARC’, ‘RIGHT-ARC’}
@data
‘the’, ‘cat’, ‘-’, ‘-’, ‘SHIFT’
‘cat’, ‘sat’, ‘the’, ’-’, ‘LEFT-ARC’
‘cat’, ‘sat’, ‘-’, ‘-’, ‘SHIFT’
‘sat’, ‘on’, ‘cat’, ‘-’, ‘LEFT-ARC’
‘sat’, ‘on’, ‘-’, ‘-’, ‘SHIFT’
‘on’, ‘the’, ‘sat’, ‘-’, ‘SHIFT’
‘the’, ‘mat’, ‘on’, ‘sat’, ‘SHIFT’
‘mat’, ‘-’, ‘the’, ‘on’, ‘LEFT-ARC’
‘mat’, ‘-’, ‘on’, ‘sat’, ‘RIGHT-ARC’
‘on’, ‘-’, ‘sat’, ‘-’, ‘RIGHT-ARC’
‘sat’, ‘-’, ‘-’, ‘-’, ‘SHIFT’
```

Figure 5.17: An example of data for a *.arff* file.

kind of situation where you should carry out a `RIGHT-ARC`.)


Classifier output

```
stack_word_pos_1 = ?: SHIFT (8430.0)
stack_word_pos_1 = ABBREV
|   queue_word_pos_1 = ABBREV
|   |   queue_word_pos_2 = ?: RIGHT-ARC (6.0)
|   |   queue_word_pos_2 = ABBREV
|   |   |   queue_word_pos_3 = ?: RIGHT-ARC (5.0)
|   |   |   queue_word_pos_3 = ABBREV: RIGHT-ARC (2.0)
|   |   |   queue_word_pos_3 = ADJ: RIGHT-ARC (8.0)
|   |   |   queue_word_pos_3 = ADV: RIGHT-ARC (1.0)
|   |   |   queue_word_pos_3 = CONJ: RIGHT-ARC (0.0)
|   |   |   queue_word_pos_3 = CV: RIGHT-ARC (0.0)
|   |   |   queue_word_pos_3 = DET: RIGHT-ARC (0.0)
|   |   |   queue_word_pos_3 = INTERJ: RIGHT-ARC (0.0)
|   |   |   queue_word_pos_3 = IV: RIGHT-ARC (0.0)
|   |   |   queue_word_pos_3 = LATIN: RIGHT-ARC (0.0)
|   |   |   queue_word_pos_3 = NOUN: RIGHT-ARC (3.0)
|   |   |   queue_word_pos_3 = NOUN_NUM: RIGHT-ARC (0.0
|   |   |   queue_word_pos_3 = NOUN_PROP
|   |   |   |   stack_word_pos_2 = ?: SHIFT (0.0)
|   |   |   |   stack_word_pos_2 = ABBREV: SHIFT (0.0)
|   |   |   |   stack_word_pos_2 = ADJ: SHIFT (0.0)
|   |   |   |   stack_word_pos_2 = ADV: SHIFT (0.0)
```

Figure 5.18: A screenshot of the J48 classifier output using WEKA.

4. Generating a parse model from the classification output: Finally, we convert the output produced by the classification algorithm (such as the one shown in Figure 5.18) to an appropriate question-answer model that we can use for guiding the parser to parse new sentences. Figure 5.19 is a sample of some questions and answers we have extracted from the classification output from Figure 5.18.

```
 question(
 QUEUE,
 STACK,
      [
       word_pos(STACK, 1, '-'), 'SHIFT',
       word_pos(STACK, 1, 'ABBREV'),
          [
           word_pos(QUEUE, 1, 'ABBREV'),
              [
               word_pos(QUEUE, 2, '-'), 'RIGHT-ARC',
               word_pos(QUEUE, 2, 'ABBREV'),
                  [
                   word_pos(QUEUE, 3, '-'), 'RIGHT-ARC',
                   word_pos(QUEUE, 3, 'ABBREV'), 'RIGHT-ARC',
                   word_pos(QUEUE, 3, 'ADJ'), 'RIGHT-ARC',
                   ...
                    word_pos(QUEUE, 3, 'NOUN_PROP'),
                       [
                        word_pos(STACK, 2, '-'), 'SHIFT',
                        word_pos(STACK, 2, 'ABBREV'), 'SHIFT',
                        ...
                        ]
                   ]
               ]
           ]
       ]
).
```

Figure 5.19: An example of a question-answer model.

5.3 Classification-driven deterministic parsing

It is possible to run DNDParser deterministically in different ways. If the parser is presented with a state that has one or more items on the queue but an empty stack then it will produce one state by performing SHIFT. For example, having a queue with `[1,2,3,4]` and an empty stack `[]` then the parser cannot recommend a RIGHT-ARC or a LEFT-ARC operation because either of these two operations requires an item from the stack to be made the parent or the daughter of the head of the queue respectively.

Having one or more items on the queue and one item on the stack the parser produces three states, namely: SHIFT, RIGHT-ARC, and LEFT-ARC. In this kind of situation, the parsing rules recommend only one operation where we give it a positive score so that the parser can then explore the recommended operation. However, it is possible that the classification rules may not recommend any operations if they are presented with a situation that has never been seen during training. This is possible because the classifier may not learn what action to take in every situation the parser encounters during the testing phase. For example, in Figure 5.20 we assume that the classification rules did not recommend any operation, where all three operations receive a score of 0, and thus they will all have equal scores (which will be the score

```
States         | Action      Queue       Stack    Score
---------------|-----------------------------------------
Current state  | θ           [2,3,4]     [1]      0
New states     | SHIFT       [3,4]       [2, 1]   0
               | RIGHT-ARC   [1,3,4]     []       0
               | LEFT-ARC    [2,3,4]     []       0
```

Figure 5.20: Generating three parse states from one state.

inherited from the original state). In this kind of situation, it is not clear which operation the parser should explore first, LEFT-ARC, RIGHT-ARC or SHIFT. There are six different orders-of-preference we can give to the parser as to which operation it should explore first. These are:

1. `SHIFT, LEFT-ARC, RIGHT-ARC (SHIFT-LA-RA)`
2. `SHIFT, RIGHT-ARC, LEFT-ARC (SHIFT-RA-LA)`
3. `LEFT-ARC, SHIFT, RIGHT-ARC (LA-SHIFT-RA)`
4. `LEFT-ARC, RIGHT-ARC, SHIFT (LA-RA-SHIFT)`
5. `RIGHT-ARC, SHIFT, LEFT-ARC (RA-SHIFT-LA)`
6. `RIGHT-ARC, LEFT-ARC, SHIFT (RA-LA-SHIFT)`

Furthermore, in situations where the parser is presented with a state that has one or more items on the queue and more than one item on the stack, the parser can then generate more than three states because it checks for relations between the head of the queue and any items on the stack; i.e., states that are generated by LEFT-ARC(*N+1*) and RIGHT-ARC(*N+1*). In this kind of situation, it is possible that two or more states may be recommended by the parsing rules, where two or more states receive positive scores. For example, in Figure 5.21 where the parsing rules suggested LEFT-ARC(*1*) (making 3 from the queue the parent of 2 on the stack) and also LEFT-ARC(*2*) (making 3 the head of the queue the parent of 1 from the stack) they are both given a score of 1. In this kind of exemplified situation we may deterministically choose to perform LEFT-ARC(*1*) instead of LEFT-ARC(*2*), by giving more priority to reduce operations that involve two items that are closer to each other. Alternatively, we may deterministically choose LEFT-ARC(*2*), by giving priority to reduce operations that involve two items that are further away from each other. This leads to another two different parsing variations, which are:

1. furthest-item-first: This operation involves making relations between the head of the queue with an item that is furthest away from it on the stack.

2. closest-item-first: This operation involves making relations between the head of the queue with an item on the stack that is closest to it.

```
States         | Action          Queue       Stack        Tree      Parsing rules score
---------------|-----------------------------------------------------------------------
Current state  | θ               [3,4]       [2,1]        θ         0
New states     | SHIFT           [4]         [3,2,1]      θ         0
               | RIGHT-ARC(1)    [2,4]       [1]          (2>3)     0
               | RIGHT-ARC(2)    [1,2,4]     []           (1>3)     0
               | LEFT-ARC(1)     [3,4]       [1]          (3>2)     1
               | LEFT-ARC(2)     [2,3,4]     []           (3>1)     1
```

Figure 5.21: Generating more than three parse states from one state.

Deterministic parsers accept the final parse state by assuming that the parser generated the required parse analysis. The DNDParser assumes that the final state is generated if the queue is empty, which indicates that the sentence is processed because all the words would have been inspected by the parser. The deterministic version of DNDParser accepts the final parse state when it is presented with it the first time, even if a complete analysis for a given sentence is not produced (i.e., there may be several items with no head).

5.4 Classification-driven non-deterministic parsing

In deterministic parsing, DNDParser accepts the final state as soon as it is produced; i.e., when the queue becomes empty because processing of all the words in it is performed by removing queue items on to the stack. The parser then assumes that parsing has completed. However, if we run the parser non-deterministically, we can allow it to explore the alternative states that will remain on the agenda if the final state does not contain a complete parse tree; i.e., where the stack has more than one item on it, which means that some words did not receive a parent and hence a complete parse tree is not produced. This means that the parser rolls back to the previous highest scored state on the agenda and explores it until a state is generated whereby the queue is empty and the stack contains one item whose span covers the entire length of the sentence.

Additionally, the original algorithm (the arc-standard of MaltParser) deterministically follows the reduce operations that involve the head of the queue with the top item on the stack. However, DNDParser allows the reduce operations to combine the head of the queue with any item on the stack but when the parser is presented with multiple equally scored states then it deterministically chooses a state, as described in the previous section.

5.4.1 Labelled attachment score

In this section we briefly explain how we obtain labelled attachment scores. For each dependency relation between two words, a syntactic label is attached to indicate the syntactic role of the dependent item with its head. In order to assign the correct labels to parse trees during parsing, we have extracted different patterns from the training data during parser training.

Figure 5.22 shows some examples of the extracted patterns. Here, a window size of eight items is used for items collected from the queue and the stack during training phase. In order to identify what kind of pattern produces the greatest accuracy, we have identified three different types of patterns:

Pattern one: The first element of this pattern, as shown in Figure 5.22, is the head. The second element is a list of up to eight items collected from the queue and the stack during training. The third element is the dependent item. The fourth element is the label of the dependent item. The first line in Figure 5.22 shows `'PV'` as the head of `'NOUN'` and the label of the dependent is `'SBJ'`. The second example shows that the dependent item (`'NOUN'`) is the object of the head item (`'PV'`).

```
PV, [NOUN, PV, ICONJ, PRON, PREP, NOUN, ADJ, CONJ], NOUN, SBJ
PV, [PV, NOUN, PREP, PV, ICONJ, NOUN, PREP, NOUN], NOUN, OBJ
```

Figure 5.22: Examples of pattern one for assigning labels to dependency relations.

Pattern two: The first element of this pattern, as shown in Figure 5.23, is the head. The second element is a list of up to eight items collected from the queue and the stack during training. The third element is the dependent item. The fourth element is the label of the dependent item and the last element is the distance between the head item and the dependent item. The first example in Figure 5.23 shows that the label of the dependent is `'SBJ'` whereby `'PV'` is the head of `'NOUN'` and whether they are adjacent because the distance between them is 1. The second example shows that the label of the dependent is `'OBJ'` whereby `'PV'` is the head of `'NOUN'` and the distance between the head and the dependent is 2.

PV, [NOUN, PV, ICONJ, PRON, PREP, NOUN, ADJ, CONJ], NOUN, SBJ, 1.
PV, [PV, NOUN, PREP, PV, ICONJ, NOUN, PREP, NOUN], NOUN, OBJ, 2.

Figure 5.23: Examples of pattern two for assigning labels to dependency relations.

Pattern three: The first element of this pattern, as shown in Figure 5.24, is the head. The second element is a list of up to eight items collected from the queue and the stack during training. The third element is the dependent item. The fourth element is the label of the dependent item and the last element is the frequency of the pattern recorded during training, which is used for computing the probability of the pattern during parsing. The first example in Figure 5.24 shows that the `'PV'` is the head of `'NOUN'`, the label of the dependent is `SBJ`, and that the last element indicates that the pattern occurred 6 times during training. The second example shows that the dependent item `'NOUN'` is the object of the head item (`'PV'`) which occurred 4 times during training.

PV, [NOUN, PV, ICONJ, PRON, PREP, NOUN, ADJ, CONJ], NOUN, SBJ, 6.
PV, [PV, NOUN, PREP, PV, ICONJ, NOUN, PREP, NOUN], NOUN, OBJ, 4.

Figure 5.24: Examples of pattern three for assigning labels to dependency relations.

The assignment of labels to dependency relations is performed in two steps: (i) the dependency relation between two words is established, and (ii) one of the patterns above is used for determining what label should be assigned to the dependency relation. This approach differs from the approach used in MaltParser. MaltParser learns models during the training phase for assigning labels to dependency relations when it performs LEFT-ARC or RIGHT-ARC reduce operations. Alternatively, it is possible to train two (or more) separate models where the first model determines the unlabelled attachments dependency relations while the second model determines the labels for the dependency relations between two words during the reduce operations[17]. We show in our experiments in the following chapter that our approach to label assignment for dependency relations yields a better accuracy result than MaltParser.

[17]This information is obtained via email communications with Joakim Nivre and Johan Hall on 31 January 2015.

5.5 Summary

In this chapter we have described the relationship between phrase structure trees and dependency trees. And we have presented an approach for converting phrase structure trees to dependency formats. We have conducted several experiments on MaltParser using different data-sets, where each dataset was based on a version of HPT that was used for converting the dataset from phrase structure trees to dependency trees.

Moreover, we have presented a small modification to the arc-standard algorithm of MaltParser and we have shown that it is possible to process non-projective sentences (this has been demonstrated in Section 5.2.1). We have presented a technique for assigning scores to different parse states where these states are stored in an agenda. The agenda is stored in a chart table. In this way we have used features of shift-reduce parsing with chart parsing. Furthermore, we have demonstrated the possibility of running our parser deterministically or non-deterministically and have shown that non-deterministic parsing performed better than deterministic parsing.

Finally, we have described the steps in generating a parse model and we have outlined our approach to assigning labels to dependency relations.

Chapter 6

Parser Evaluation

In this chapter, we will evaluate our parsing algorithm. Firstly, we will evaluate a large a number of machine learning algorithms in order to identify those algorithms that produce good classification accuracy, because we believe classifying our data correctly will help in improving the accuracy of the parser. Secondly, we will evaluate our parser by training it on those classifiers with 80% accuracy or more in order to test the effect of each classifier on parsing accuracy. In Section 6.5.2 we will describe the way we extract different types of constraint rules from our dependency treebank and integrate them in our parser. Finally, we will compare the accuracy of our parser with that of the arc-standard algorithm of MaltParser.

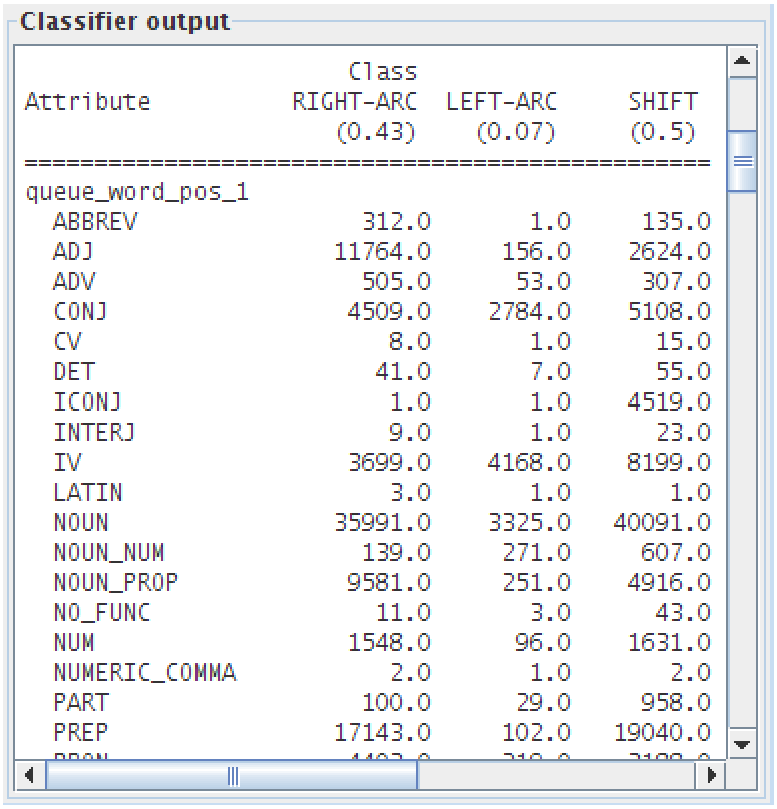
6.1 The evaluation of various classification algorithms

Table 6.1 contains the results for a number of machine learning algorithms that we have used for classifying the first 4000 sentences from the DATB (this is the dependency treebank that we have obtained by converting the phrase structure trees from the PATB).

We consider a machine learning algorithm appropriate for producing a parse model if it meets two requirements: (i) it produces a good classification accuracy. Although the accuracy of a classification algorithm may not directly reflect the accuracy of a parser that uses its recommendations, a classifier that produces a high level of accuracy is more likely to assist a parser to make more informed parse decisions at each parse step than a classifier that produces a low level of accuracy. (ii) its output can be used for generating a parse model (a set of questions and answers) which can be used for making recommendations to a data-driven parser; for example, what action (SHIFT, LEFT-ARC, or RIGHT-ARC) the parser should take in a specific situation.

We have used various features for training the different classification algorithms. These features include POS tags, word forms, word locations in sentences, their spans (i.e., their start and end positions in sentences). Additionally, we have used a combination of these features such as word forms with POS tags, words forms with word location or word spans, and similar combination of POS tags with other features. The use of these features for training each classification along with the classification accuracy is presented in Table 6.1.

The output of some other classifiers, such as the output of the NaiveBayes classifier shown in Figure 6.1, is opaque and therefore we cannot derive a question-answer-type parse model from it. The classifiers that have been used in subsequent sections for evaluating their effectiveness in parsing natural languages are based on those that their output allowed for deriving a usable parse model.



```
                             Class
Attribute                RIGHT-ARC  LEFT-ARC     SHIFT
                            (0.43)    (0.07)     (0.5)
==================================================
queue_word_pos_1
  ABBREV                    312.0       1.0      135.0
  ADJ                     11764.0     156.0     2624.0
  ADV                       505.0      53.0      307.0
  CONJ                     4509.0    2784.0     5108.0
  CV                          8.0       1.0       15.0
  DET                        41.0       7.0       55.0
  ICONJ                       1.0       1.0     4519.0
  INTERJ                      9.0       1.0       23.0
  IV                       3699.0    4168.0     8199.0
  LATIN                       3.0       1.0        1.0
  NOUN                    35991.0    3325.0    40091.0
  NOUN_NUM                  139.0     271.0      607.0
  NOUN_PROP                9581.0     251.0     4916.0
  NO_FUNC                    11.0       3.0       43.0
  NUM                      1548.0      96.0     1631.0
  NUMERIC_COMMA               2.0       1.0        2.0
  PART                      100.0      29.0      958.0
  PREP                    17143.0     102.0    19040.0
```

Figure 6.1: A screenshot of NaiveBayes classifier output using WEKA.

During the evaluation of the classification algorithms, some of the widely used classifiers did not yield encouraging results. For example, the LiBSVM classifier which is used in MaltParser did not perform well with the set of features that we have supplied.

It only managed to learn successfully from one feature (POS tags), while the accuracy was well below the accuracy of some of the other classifiers. The entries for LiBSVM in Table 6.1 are incomplete because training takes so long (3 days per case) that future experiments seemed infeasible. However, the fact that it produces no better classification accuracy than the J48 algorithm in the cases that we have looked at suggests that it is unlikely to substantially outperform it in the remaining cases.

From the large number of experiments we have conducted using several machine learning algorithms, we will evaluate DNDParser by training it using the classification algorithms that produced a high classification accuracy in the following section[1].

[1] All the experiments conducted in this thesis are performed on a Linux based machine with 3.10GHz processor and 4GB RAM.

J48

Items on Queue	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4
Items on Stack	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Word (%)	52.38	54.87	55.28	55.43	67.70	68.08	68.12	68.24	67.81	68.29	68.37	68.53	68.10	68.56	68.67	**68.81**
Word + location (%)	56.60	55.90	54.53	53.41	71.95	72.08	71.79	71.92	72.03	72.23	71.88	71.67	72.10	**72.41**	72.11	71.80
Word + location + span (%)	67.52	67.45	66.79	68.83	72.52	72.65	72.08	71.73	72.50	72.76	72.25	72.00	72.49	**72.87**	72.45	72.17
Word + span (%)	64.40	64.90	64.35	63.81	70.54	70.67	70.46	70.17	70.66	70.81	70.64	70.43	70.68	**70.83**	70.79	70.56
POS (%)	63.21	64.45	64.50	64.50	83.61	84.76	84.88	84.94	84.48	85.63	85.77	85.80	84.75	85.89	**86.05**	86.04
POS + location (%)	63.79	64.59	64.65	64.68	84.46	85.27	85.27	85.27	85.04	85.89	85.91	85.92	85.25	**86.96**	86.08	86.09
POS + location + span (%)	76.77	77.28	77.21	77.23	84.54	85.12	85.11	85.23	85.10	85.81	85.84	85.92	85.28	85.95	85.97	**85.99**
POS + span (%)	76.15	76.74	76.66	76.68	84.29	84.98	85.00	85.00	84.93	85.71	85.69	85.67	85.09	**85.88**	**85.88**	**85.88**
Word + POS (%)	63.31	64.40	64.20	63.87	84.07	85.25	85.25	85.28	85.05	86.23	86.24	86.24	85.25	86.46	**86.47**	86.39
Word + POS + location (%)	63.44	63.15	62.71	62.58	85.24	85.93	85.93	85.83	85.77	86.57	86.53	86.45	85.84	**86.63**	86.57	86.54
Word + POS + location + span (%)	77.54	77.82	77.54	77.45	85.40	85.90	85.84	85.83	85.89	86.48	86.54	86.47	85.94	86.49	**86.55**	86.44
Word + POS + span (%)	77.21	77.32	77.16	76.99	85.34	85.87	85.76	85.79	85.89	86.50	86.45	86.43	85.90	**86.53**	86.49	86.36

LibSVM

Items on Queue	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4
Items on Stack	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Word (%)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Word + location (%)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Word + location + span (%)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Word + span (%)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
POS (%)	61.87	63.73	63.55	63.21	73.45	74.40	74.73	74.62	74.32	75.41	75.43	75.39	74.58	75.62	**75.63**	75.55
POS + location (%)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
POS + location + span (%)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
POS + span (%)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Word + POS (%)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Word + POS + location (%)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Word + POS + location + span (%)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Word + POS + span (%)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Id3

Items on Queue	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4
Items on Stack	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Word (%)	52.36	54.75	55.01	55.02	67.62	67.85	67.71	67.65	67.64	67.94	67.77	67.68	67.74	**67.97**	67.79	67.62
Word + location (%)	54.88	51.29	47.63	44.77	**68.70**	66.02	63.63	62.37	67.83	65.22	63.04	61.89	66.92	64.41	62.55	61.49
Word + location + span (%)	63.27	60.33	58.23	56.83	**68.61**	65.65	63.69	62.69	67.77	64.85	63.18	62.26	66.89	64.23	62.79	62.00
Word + span (%)	61.77	58.91	56.36	54.74	**67.08**	64.46	62.42	61.21	66.14	63.60	61.84	60.71	65.24	62.81	61.25	60.36
POS (%)	63.21	64.07	62.78	60.76	83.56	84.15	83.04	81.64	**83.74**	83.41	81.78	80.54	82.31	81.47	79.70	78.81
POS + location (%)	57.53	50.87	46.46	44.84	**77.55**	75.79	74.80	74.57	76.25	75.48	75.04	74.95	74.94	74.83	74.65	74.61
POS + location + span (%)	71.96	68.77	66.95	66.27	**77.57**	75.71	74.70	74.51	76.31	75.42	74.92	74.83	75.13	74.85	74.63	74.57
POS + span (%)	71.72	68.43	66.45	65.69	**77.42**	75.51	74.49	74.28	76.10	75.17	74.71	74.59	74.92	74.64	74.41	74.33
Word + POS (%)	63.05	63.17	61.18	58.68	83.46	**83.62**	82.25	81.02	83.34	82.89	81.29	80.36	81.76	81.04	79.67	79.09
Word + POS + location (%)	56.27	49.34	45.50	44.19	**77.36**	75.84	75.13	74.96	76.31	75.73	75.39	75.33	75.21	75.29	75.07	75.04
Word + POS + location + span (%)	71.76	68.65	67.01	66.48	**77.38**	75.72	74.92	74.79	76.39	75.64	75.22	75.15	75.36	75.21	75.00	74.93
Word + POS + span (%)	71.51	68.26	66.50	65.90	**77.23**	75.54	74.68	74.55	76.17	75.45	75.04	74.97	75.18	75.04	74.81	74.73

RandomTree

Items on Queue	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4
Items on Stack	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Word (%)	52.37	54.77	55.06	55.08	67.72	68.04	68.00	68.00	67.82	68.27	68.26	68.32	68.01	68.47	**68.51**	68.50
Word + location (%)	55.27	52.23	49.25	47.03	**71.36**	70.67	69.48	70.25	71.32	70.64	69.35	68.67	71.05	70.19	69.36	69.83
Word + location + span (%)	66.44	64.02	-	-	**72.02**	70.55	-	-	71.83	70.27	-	-	71.50	69.937	-	-
Word + span (%)	64.04	63.25	61.18	59.71	**69.97**	69.27	68.23	67.46	69.80	68.99	68.25	67.39	69.63	68.67	67.79	67.89
POS (%)	63.21	64.17	63.12	61.45	83.67	84.82	84.50	83.71	84.47	**85.28**	84.71	84.26	84.26	84.78	84.31	83.69
POS + location (%)	58.73	53.11	49.39	47.72	**83.04**	82.93	80.76	79.18	82.77	81.41	80.15	78.84	81.71	80.09	78.18	80.12
POS + location + span (%)	76.02	70.97	69.12	69.12	**82.54**	81.00	79.17	76.32	81.78	79.41	78.02	76.26	80.27	78.79	77.42	76.47
POS + span (%)	75.68	73.35	68.10	67.72	**83.15**	81.47	79.89	79.19	81.50	80.89	78.69	77.78	80.60	79.93	77.28	76.95
Word + POS (%)	63.13	63.40	61.70	59.46	84.08	**85.02**	84.34	83.62	84.80	85.37	84.34	83.57	84.36	84.46	83.33	83.28
Word + POS + location (%)	57.51	51.66	48.15	46.84	**83.33**	82.06	81.23	80.10	82.02	81.84	80.62	79.17	81.24	80.27	79.03	77.99
Word + POS + location + span (%)	75.44	72.24	-	-	**82.91**	80.68	78.92	77.35	81.75	79.59	77.50	76.33	80.48	78.09	76.77	75.02
Word + POS + span (%)	75.68	71.56	68.87	-	**82.89**	81.75	79.46	78.42	81.71	80.58	77.87	77.34	80.37	78.95	76.87	77.17

NaiveBayes

Items on Queue	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4
Items on Stack	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Word (%)	51.89	53.98	54.10	54.58	66.71	**66.99**	61.83	60.13	65.70	65.95	65.48	63.37	64.57	65.06	64.68	64.60
Word + location (%)	53.22	53.69	53.98	53.56	66.20	60.72	57.56	57.02	**64.45**	64.12	57.03	55.68	62.10	62.21	54.67	52.74
Word + location + span (%)	53.45	51.97	48.89	48.79	**60.05**	50.95	47.39	49.98	55.98	47.29	44.51	47.60	51.12	45.28	42.79	45.28
Word + span (%)	53.46	53.42	52.31	51.08	**63.03**	56.69	51.92	53.47	61.93	55.32	48.75	51.09	58.75	52.30	46.46	48.57
POS (%)	61.87	61.46	60.12	59.39	**78.36**	76.58	74.79	70.42	76.95	76.78	76.19	74.02	75.40	76.01	75.38	74.17
POS + location (%)	60.05	59.12	58.17	58.31	**77.46**	73.49	66.60	64.05	74.76	74.70	71.13	67.13	71.35	72.39	70.68	67.1
POS + location + span (%)	58.37	57.68	56.00	55.30	**72.95**	65.37	59.14	58.43	69.06	65.72	58.22	57.11	63.75	61.27	55.49	54.24
POS + span (%)	59.77	58.54	57.37	57.24	**75.08**	69.93	62.92	61.15	71.68	70.93	65.13	61.31	67.19	67.62	63.37	59.63
Word + POS (%)	59.38	58.67	58.20	58.13	76.55	**76.72**	71.35	66.00	73.84	74.67	73.66	71.04	70.76	72.12	72.21	71.02
Word + POS + location (%)	58.54	57.84	57.39	57.33	**76.23**	73.52	64.61	62.25	71.93	72.50	69.20	63.57	68.08	69.13	67.89	62.70
Word + POS + location + span (%)	57.79	56.92	55.66	54.77	**72.86**	65.40	58.09	57.62	67.91	64.58	56.72	55.55	62.92	59.78	53.95	52.73
Word + POS + span (%)	58.78	57.51	56.76	56.25	**74.54**	69.66	61.49	59.98	69.87	69.69	62.84	59.08	65.67	65.33	60.50	56.89

DecisionStump

Items on Queue	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4
Items on Stack	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Word (%)	50.00	50.00	50.00	50.00	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77
Word + location (%)	50.00	50.00	50.00	50.00	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77
Word + location + span (%)	50.00	50.00	50.00	50.00	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77
Word + span (%)	50.00	50.00	50.00	50.00	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77
POS (%)	50	50	50	50	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58
POS + location (%)	50	50	50	50	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58
POS + location + span (%)	50	50	50	50	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58
POS + span (%)	50.00	50.00	50.00	50.00	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58	60.58
Word + POS (%)	50.00	50.00	50.00	50.00	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77
Word + POS + location (%)	50.00	50.00	50.00	50.00	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77
Word + POS + location + span (%)	50.00	50.00	50.00	50.00	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77
Word + POS + span (%)	50.00	50.00	50.00	50.00	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77	63.77

Table 6.1: Classification accuracy with various items on the queue and stack.

6.2 Optimising unlabelled attachment

In this section we will evaluate DNDParser using deterministic and non-deterministic approaches. Our aim here is to check the parsing accuracy for unlabelled attachment scores (UAS) and its speed (the time taken for determining each dependency relation between two words in seconds). All of the testings conducted in this section are based on 1-fold validation because our goal is to find the best parsing strategy with a specific classifier. Carrying out 5-fold validation would simply make the experiments take five times as long without producing much extra information. However, in Section 6.4.2 we conduct 5-fold cross validations.

6.2.1 Deterministic parsing evaluation

Here we use the classification algorithm that has the highest accuracy to drive the parsing algorithm completely deterministically. We will evaluate the deterministic parsing algorithm using the different variations described in Section 5.3. In these experiments, the parser must accept the final state that it produces, which may or may not contain a complete parse tree, since it is possible that the final state (where the queue is empty; i.e., the parser processed all the words in the queue) produced by the parser may not be a terminal state; i.e., the queue is empty but the stack has more than one item, meaning that all the items on the queue have been processed but a full parse tree has not been produced.

We have evaluated DNDParser deterministically to observe its performance when using different orders-of-preference when the parser is presented with a number of equally scored states. We have used the setting POS+LOC feature with a window size of four items on the queue and two items on the stack because this gives the highest classification accuracy. The result of deterministic parser evaluation is presented in Table 6.2.

The parser does not perform well when it is used deterministically, but it is noticeable that performing furthest-item-first reduction is a better strategy to use than closest-item-first reduction. Moreover, in both strategies, the LA-SHIFT-RA orders-of-preference (from page 122) produces better parsing accuracy than the other orders.

The results presented in Table 6.2 indicate that running the parser with different strategies yields different results. We anticipate that the parser does not perform well when the parser accepts the first established final parse state as the final analysis. We believe that the parsing performance may improve if the parser is allowed to roll back to the previous state that has the highest score in the agenda when a final parse state does not contain a complete parse analysis for a given sentence. This non-deterministic approach is explored in the following section.

Furthest-item-first reduction						
Classification algorithm	Feature	Queue size	Stack size	orders-of-preference	UAS (%)	Speed
J48	POS+LOC	4	2	**LA-SHIFT-RA**	**52.27**	**0.054**
				SHIFT-RA-LA	45.93	0.052
				SHIFT-LA-RA	46.02	0.046
				RA-SHIFT-LA	28.33	0.034
				RA-LA-SHIFT	28.33	0.033
				LA-RA-SHIFT	28.33	0.035

Closest-item-first reduction						
Classification algorithm	Feature	Queue size	Stack size	orders-of-preference	UAS (%)	Speed
J48	POS+LOC	4	2	**LA-SHIFT-RA**	**46.12**	**0.046**
				SHIFT-LA-RA	45.62	0.046
				SHIFT-RA-LA	46.0	0.112
				RA-SHIFT-LA	28.31	0.033
				RA-LA-SHIFT	27.89	0.028
				LA-RA-SHIFT	27.89	0.029

Table 6.2: Deterministic parsing with different strategies and POS tags and location as features and a window size of four items on the queue and two items on the stack.

6.2.2 Non-deterministic parsing evaluation

In this section we use the same classification algorithm that we have used for evaluating DNDParser deterministically. Here, we evaluate it non-deterministically using the different variations described in Section 5.3. In these experiments, the parser is allowed to roll back to the previous highest scored state if the final state does not contain a complete parse tree.

The results for non-deterministic parsing evaluation are presented in Table 6.3. Among all of the experiments in this section, as shown in Table 6.3 (and in the previous section), it is apparent that the furthest-item-first reduction strategy performs better than the other strategy (closest-item-first reduction). However, the SHIFT-LA-RA orders-of-preference produces better results when DNDParser was used non-deterministically. The reason that LA-SHIFT-RA order produced better results when DNDParser was used deterministically is because the parser performs LEFT-ARC instead of SHIFT when the classifier fails to recommend an operation, which results in assigning an item on the stack as the parent of an item on the queue. If a SHIFT operation is performed instead, then the parser would produce a less complete tree when the final state is reached the first time, whereas a less complete tree would mean more errors

Furthest-item-first reduction						
Classification algorithm	Feature	Queue size	Stack size	orders-of-preference	UAS (%)	Speed
J48	POS+LOC	4	2	**SHIFT-LA-RA**	**70.6**	**0.074**
				LA-SHIFT-RA	52.39	0.062
				SHIFT-RA-LA	44.77	0.066
				RA-LA-SHIFT	30.02	0.052
				RA-SHIFT-LA	28.31	0.043
				LA-RA-SHIFT	28.31	0.052

Closest-item-first reduction						
Classification algorithm	Feature	Queue size	Stack size	orders-of-preference	UAS (%)	Speed
J48	POs+LOC	4	2	**SHIFT-LA-RA**	**62.6**	**0.079**
				LA-SHIFT-RA	45.63	0.050
				SHIFT-RA-LA	45.12	0.177
				RA-LA-SHIFT	27.88	0.052
				LA-RA-SHIFT	27.88	0.041
				RA-SHIFT-LA	28.29	0.043

Table 6.3: Non-deterministic parsing with different strategies and POS and location as features with a window size of four items on the queue and two items on the stack.

in the parse analyses and hence lower accuracy. Also, performing LEFT-ARC assigns stack items as the parent of the head of the queue at the earliest possible stages where the operation should not have been performed.

In the case of non-deterministic parsing, DNDParser benefits from the SHIFT-LA-RA order because the parser has the chance to explore different states if the queue becomes empty and a complete parse tree is not produced. When the classifier fails to recommend a parse operation, then it is safer to perform a SHIFT operation because the parser can roll back to the previous highest scored state if the SHIFT does not lead to a complete parse analysis.

The results shown in Tables 6.2 and 6.3 indicate that running DNDParser non-deterministically is more effective than running it deterministically in terms of accuracy. However, running the parser non-deterministically is about 50% slower than running it deterministically.

In the following section, we will repeat the experiments in Sections 6.2.1 and 6.2.2 by training the parser on the J48 classifier but with different features and settings. Our goal in repeating those experiments is to find an answer to the following question: “Will the parser perform better or worse if it is trained with the same classification algorithm but with a different set of features and settings?” We also present a number of experiments by running the parser trained on a number of classification algorithms with various features and settings.

6.3 Parser evaluation with different features and settings

In this section we will evaluate DNDParser in the same way as in the previous section. However, in this section, the parser is trained using different settings and features. Here, we train it with J48 but with POS tags only as features and window sizes of four item on the queue and three items on the stack. Our goal in conducting these evaluations is to find out whether the parser performs differently if it is trained with different features and settings. The results of our findings are presented in Tables 6.4 and 6.5.

Furthest-item-first reduction						
Classification algorithm	Feature	Queue size	Stack size	orders-of-preference	UAS (%)	Speed
J48	POS	4	3	**LA-SHIFT-RA**	**72.48**	**0.062**
				SHIFT-LA-RA	59.77	0.047
				SHIFT-RA-LA	59.41	0.067
				RA-LA-SHIFT	53.67	0.043
				LA-RA-SHIFT	53.67	0.042
				RA-SHIFT-LA	53.67	0.041

Closest-item-first reduction						
Classification algorithm	Feature	Queue size	Stack size	orders-of-preference	UAS (%)	Speed
J48	POS	4	3	**LA-SHIFT-RA**	**66.46**	**0.058**
				SHIFT-LA-RA	59.76	0.035
				SHIFT-RA-LA	59.58	0.41
				RA-SHIFT-LA	52.62	0.037
				RA-LA-SHIFT	51.35	0.030
				LA-RA-SHIFT	51.35	0.032

Table 6.4: Different strategies for deterministic parsing with POS only as features with a window size of four items on the queue and three items on the stack.

Furthest-item-first reduction						
Classification algorithm	Feature	Queue size	Stack size	orders-of-preference	UAS (%)	Speed
J48	POS	4	3	**SHIFT-LA-RA**	**76.12**	**0.074**
				LA-SHIFT-RA	72.61	0.062
				SHIFT-RA-LA	57.54	0.078
				RA-SHIFT-LA	53.6	0.060
				RA-LA-SHIFT	53.6	0.059
				LA-RA-SHIFT	53.6	0.060

Closest-item-first reduction						
Classification algorithm	Feature	Queue size	Stack size	orders-of-preference	UAS (%)	Speed
J48	POS	4	3	**SHIFT-LA-RA**	**70.75**	**0.076**
				LA-SHIFT-RA	66.48	0.058
				SHIFT-RA-LA	57.01	0.077
				RA-SHIFT-LA	52.55	0.056
				LA-RA-SHIFT	51.34	0.052
				RA-LA-SHIFT	51.34	0.051

Table 6.5: Non-deterministic parsing evaluation with different strategies and POS only as features with a window size of four items on the queue and three items on the stack.

It appears that using different settings affects the performance of the parser greatly.

From the experiments conducted in this section, and the previous section, it is apparent that running the parser non-deterministically with `SHIFT-LA-RA` orders-of-preference and using a furthest-item-first reduction strategy produces the best parsing performance. In the following section we will conduct a large number of experiments by training the parser on a number of different classifiers with different features and settings.

6.4 Evaluating classifiers for parsing

In the previous section (Section 6.1) a number of classification algorithms were identified. As presented in Table 6.1 the classification accuracy varies because each algorithm learns differently from the set of training data. In this section, we investigate the effect of using different classification algorithms on the parser's performance when it is being trained by different classifiers. Specifically, we train our parser by using those classifiers that produce the highest classification accuracy with a `SHIFT-LA-RA` orders-of-preference and a furthest-items-first reduction strategy. The objective in using different classifiers for training the parser is to identify the algorithms that help the parser perform best in terms of accuracy and efficiency.

These experiments also highlight whether generating different parsing models by using different classification algorithms contribute in different ways to parsing performance. The optimal accuracy of a classification accuracy may or may not necessarily lead to better parsing performance. Hence, we can investigate the effectiveness of different classification algorithms in parsing natural languages.

From Table 6.1, we can identify the classification algorithms with the highest degree of accuracy. In this section, we train DNDParser using J48, RandomTree, and Id3 algorithms since they all classified the same set of training data with over 80% accuracy. For each of these algorithms we use the same settings that produced the optimal accuracy (see the numbers in bold with grey background in Table 6.1). For example, based on the results in Table 6.1, we will use the POS tags as a training feature for the J48 algorithm with four items on the queue and four items on the stack because with this setting the algorithm produced 86.05% accuracy, while if we are using POS tags and their locations in a sentence as training features for the J48 algorithm then we will use four items on the queue and two items on the stack because the algorithm performs best with this setting, which produced 86.96% accuracy. Moreover, we limit the window size for the queue and stack to a maximum of 4 items because previous experiments conducted by Jaf and Ramsay [Jaf and Ramsay, 2013] had shown that using a larger window size than 4 items did not have any effect on the performance of parsing accuracy. For the experiment results presented in Table 6.6, we have used DNDParser non-deterministically, with the SHIFT-LA-RA orders-of-preference and a furthest-item-first reduction strategy because the parser performed best with this setting.

Classifier	Features	Classification Accuracy	Queue size	Stack size	UAS (%)	Speed
J48	**POS**	**86.05%**	**4**	**3**	**76.1**	**0.074**
J48	POS	86.04%	4	4	75.4	0.080
J48	POS + location	86.96%	4	2	70.3	0.146
J48	POS + location + span	85.99%	4	4	69.2	0.161
J48	POS + span	85.88%	4	2	70.6	0.145
J48	POS + span	85.88%	4	3	70.8	0.150
J48	POS + span	85.88%	4	4	70.9	0.142
J48	Words + POS	86.47%	4	3	71.4	0.096
J48	Words + POS + location	86.63%	4	2	69.9	0.140
J48	Words + POS + location + span	86.55%	4	3	68.0	0.183
J48	Words + POS + span	86.53%	4	2	69.8	0.161
RandomTree	POS	85.28%	3	2	70.9	0.141
RandomTree	POS + Location	83.04%	2	1	68.6	0.154
RandomTree	POS + Location + span	82.54%	2	1	67.8	0.181
RandomTree	POS + span	83.15%	2	1	68.2	0.181
RandomTree	Words + POS	85.02%	2	2	70.0	0.196
RandomTree	Words + POS + location	83.33%	2	1	68.7	0.198
RandomTree	Words + POS + location + span	82.91%	2	1	66.8	0.196
RandomTree	Words + POS + span	82.89%	2	1	68.6	0.184
Id3	POS	83.74%	3	1	70.6	0.083
Id3	Words + POS	83.62%	2	2	68.1	0.099

Table 6.6: Non-deterministic parsing with different classification algorithms, features and settings.

The results of the evaluation of our parser are presented in Table 6.6. The best parsing performance is achieved when training the parser using the J48 classification algorithm on only POS tags as a feature and the window size for the queue and stack is four and three respectively (marked bold in Table 6.6). The experiments in Table 6.6 show that training a classifier using a small set of features produces relatively similar classification accuracy to using a larger set of features. However, using a smaller set of features helps the parser to produce more accurate parse analyses and this can be achieved more quickly than using a larger set of features. It can be noted from Table 6.6 that using the J48 algorithm with only POS tags as a training feature produces a lower level of classification accuracy (86.05%) than when using POS tags and their locations as training features (86.96%). However, using the former features helps the parser to produce the best parsing results. Using only POS tags improves the parsing accuracy for unlabelled attachment score by 5.8%. Additionally, using a smaller set of features makes the parser 50% quicker. Furthermore, using a smaller window size for the queue and the stack is better for parsing performance. Training the parser on the same training feature (POS tags only) but with a smaller window size (four items on the queue and three items on the stack) improves the parsing accuracy by 0.78% for unlabelled attachment score.

The reason for the improved parsing efficiency when training the parser on smaller sets of feature is that the question set that the parser searches through is smaller and hence it will find an answer more quickly compared to situations where there are several features for searching through. The reason for the improved parsing accuracy when training the parser on smaller set of features is most likely because the parser learns consistently from a smaller set of features rather than from several features where the training data is not large enough to learn more effectively.

In the following sections we will evaluate the labelled attachment score of our parser. We will also attempt to extract different kinds of constraint rules from a dependency treebank and apply them to the our data-driven parser. Our goal is to find out whether applying constraint rules to DNDParser improves the parsing performance. In the remaining experiments, we will use the J48 classifier with POS tags only as features, with window sizes of four items on the queue and three items on the stack because this classifier with this setting produced the best parsing performance.

6.4.1 Labelled attachment score evaluation

In Section 5.4.1, we have described the extraction of different kinds of patterns for determining the dependency labels for each dependency relation during parsing. In this section, we experiment with using these rules and report on the parsing performance regarding the labelled attachment accuracy. We are using the parse setting from the previous experiments that produced the optimal parsing performance; i.e., training the parser with the J48 classifier with POS tags as features and four items on the queue and three items on the stack. Furthermore, we run the parser non-deterministically with SHIFT-LA-RA orders-of-preference and furthest-item-first reduction strategy.

The tables below show the results for using each kind of pattern with a number of different window sizes of the queue and stack. From the large number of experiments, we see that using pattern two with a window size of five items on the queue and five items on the stack produces the highest labelled attachment accuracy. We will be using pattern two for the subsequent experiments when assigning labels to dependency relations.

Different window size of queue and stack (pattern one)					
Queue size	Stack size	UAS	LAS	LA	Speed
2	2	76.1	59.3	77.73	0.075
2	3	76.1	65.7	85.02	0.076
2	4	76.1	68.1	87.92	0.077
2	5	76.1	69.2	89.28	0.076
3	2	76.1	63.3	82.39	0.077
3	3	76.1	68.6	88.5	0.079
3	4	76.1	70.3	90.56	0.078
3	5	76.1	70.8	91.07	0.078
4	2	76.1	67.0	86.78	0.078
4	3	76.1	70.2	90.48	0.079
4	4	76.1	71.3	91.75	0.08
4	5	76.1	71.5	92.05	0.08
5	2	76.1	68.4	88.48	0.08
5	3	76.1	70.9	91.33	0.081
5	4	76.1	71.7	92.26	0.082
5	**5**	**76.1**	**71.9**	**92.49**	**0.082**

Table 6.7: Parser evaluation for LAS using pattern one.

Different window size of queue and stack + distance between head and dependent (pattern two)					
Queue size	Stack size	UAS	LAS	LA	Speed
2	2	76.1	67.6	87.81	0.074
2	3	76.1	69.3	89.68	0.075
2	4	76.1	70.0	90.58	0.076
2	5	76.1	70.8	91.42	0.077
3	2	76.1	68.2	88.52	0.076
3	3	76.1	70.1	90.72	0.077
3	4	76.1	70.9	91.62	0.078
3	5	76.1	71.3	91.98	0.078
4	2	76.1	69.5	90.01	0.078
4	3	76.1	70.7	91.32	0.079
4	4	76.1	71.4	92.15	0.08
4	5	76.1	71.7	92.4	0.08
5	2	76.1	70.4	91.03	0.08
5	3	76.1	71.2	91.94	0.081
5	4	76.1	71.8	92.57	0.082
5	**5**	**76.1**	**72.0**	**92.66**	**0.082**

Table 6.8: Parser evaluation for LAS using pattern two.

Different window size of queue and stack with label probability (pattern three)					
Queue size	Stack size	UAS	LAS	LA	Speed
2	2	76.1	60.7	79.15	0.074
2	3	76.1	66.7	86.12	0.075
2	4	76.1	69.2	89.14	0.076
2	5	76.1	70.4	90.49	0.077
3	2	76.1	63.9	82.94	0.076
3	3	76.1	69.0	88.94	0.077
3	4	76.1	70.8	91.08	0.078
3	5	76.1	71.3	91.58	0.078
4	2	76.1	67.3	87.08	0.078
4	3	76.1	70.5	90.75	0.08
4	4	76.1	71.5	92.01	0.08
4	5	76.1	71.8	92.31	0.08
5	2	76.1	68.4	88.56	0.08
5	3	76.1	70.9	91.4	0.081
5	4	76.1	71.7	92.34	0.081
5	**5**	**76.1**	**71.9**	**92.57**	**0.082**

Table 6.9: Parser evaluation for LAS using pattern three.

6.4.2 5-fold cross validation of DNDParser

In the previous sections we have evaluated DNDParser using 1-fold validation. Our aim for 1-fold validation was to identify the best settings, classifiers, and strategies for labelled attachment accuracy. Since we have identified the best classifier with the appropriate features and settings for parser training we will conduct 5-folds cross validation in this section and most of the following section. From the previous experiments (Section 6.2) we have identified that the parser performed best when trained on the J48 classifier with a window size of four items on the queue and three items on the stack and with POS tags only as features. Regarding the labelled attachment accuracy, we have discovered that pattern two (Section 6.4.1) leads to the highest labelled attachment score accuracy. The result for the 5-fold cross validation of the parser is presented in Table 6.10. It is worth noting that the result presented in Table 6.10 is lower than the results we have presented in the previous tables, because here we have conducted a 5-fold validation while the results in the previous tables are based on a 1-fold validation.

Classifier	Features	Classification Accuracy	Queue size	Stack size	DNDParser			
					UAS (%)	LAS (%)	LA (%)	Speed
J48	POS	86.05%	4	3	74.5	71.0	93.6	0.081

Table 6.10: 5-fold cross validation result of DNDParser.

6.5 Constraint rules and parsing

One of the extensions that we have added to the arc-standard algorithm of MaltParser is the scoring technique (see Section 5.2.2). This extension provides an opportunity to integrate constraint rules into the parser. This can be achieved by assigning a score to encourage the recommendations made by the parsing model if they produce a dependency relation that obeys the constraints. Alternatively, if we want to ignore the role of constraint rules in the parsing decision then we assign a score of zero to them. This way we deactivate the constraint rules. This corresponds to contribution C.2 in this study. Since producing a grammar for a natural language parser is an expensive and difficult task, we opt to extract different kinds of constraint rules from the dependency treebank.

Dependency data-driven parsers produce relations between different items in a given sentence based on knowledge obtained during parser training, i.e., by following recommendations made by a parse model. Parsers are trained on a set of training data by using machine learning algorithms. As shown in Table 6.1, where the accuracy of a number of machine learning algorithms is recorded, classification algorithms do not learn effectively from a set of training data completely. Thus, the parse models produced from them are prone to making recommendations to parsers that may lead to incorrect analyses, or they may fail to make any recommendations at all.

The absence of grammatical rules in data-driven parsers makes it impossible to validate the plausibility of the parse analyses produced by these parsers. Writing a set of grammatical rules for any natural languages that can be used adequately for parsing natural languages is an expensive and time consuming task. Since there is a large number of treebanks available where these treebanks contain an implicit underlying grammar, as used by the treebank annotators, one can hope to extract a set of constraint rules that are a reasonable representation of the grammar of the language in question.

Our goal in using constraint rules is to validate the parse analyses that are produced by following the recommendations made by a parsing model; i.e., to check that the recommendation made by a parsing model (for performing LEFT-ARC, or RIGHT-ARC operations) for the parser results in producing correct dependency relations, whereby the relations obey some constraint rules which are part of the language in question. In this section, we show an approach for integrating constraint rules into DNDParser, which is our contribution C.1 in this study).

6.5.1 Applying constraints to DNDParser

Integrating constraint rules into DNDParser that was presented in the previous section is fairly easy. A parse analysis that is produced by following a parse operation, which is recommended by the parse model is checked to see whether it is plausible. A parse analysis is considered plausible if it obeys the constraint rules.

A recommended parse operation is assigned a score. We attempt to use constraint rules to assign an additional score if the constraints agree with the recommendation of the parse model; i.e., if it produces a plausible parse analysis. This means that the recommendations made by a parse model are validated by using a set of constraint rules to check whether they produce acceptable parse analyses; i.e., if the parser benefits from the information provided by a parse model and from a set of constraint rules.

The effect of these rules on the parser's decision is when the parser produces a number of states from one state where each state has a score computed by adding the score given by the parsing rules to the score given by the constraint rules.

In situations where the parser is presented with a state that has one or more items on the queue and the stack has more than one item, then the parser generates more than three states because it checks for relations between the head of the queue and any item on the stack. In this kind of situation, two or more states may be given a positive score. In order to determine which of the equally scored states should be explored next, the score of the constraint rules for an operation will influence the parser's decision. For example in Figure 6.2 where the parsing rules suggested a LEFT-ARC(*1*) (making 3 from the queue the parent of 2 on the stack) and also a LEFT-ARC(*2*) (making 3 the head of the queue the parent of 1 from the stack) they were both given a score of 1. However, we assumed that the constraint rules also encouraged the recommendation of the parsing rule and that they gave their scores to the two recommended operations, where LEFT-ARC(*1*) is given as 0.25 and LEFT-ARC(*2*) is given as 0.5. In this situation, LEFT-ARC(*2*), with a total score of 1.5 plus the score of the currently explored state, will be placed on the top of the agenda because it will have the highest score. Here, in these kinds of situations, the constraint rules influence the decision of the parser when a LEFT-ARC(*2*) operation is performed instead of LEFT-ARC(*1*) operation.

Since we have a dependency treebank we can extract different types of constraint rules in the form of patterns from the dependency trees. The main type is the parent-daughter relations between the words of a sentence.

```
States         | Action         Queue       Stack       Trees    Parsing rules score  Constraint rules score   Total score
---------------|-------------------------------------------------------------------------------------------------------------
Current state  | θ              [3,4]       [2,1]       θ        0                    0                        0
New states     | SHIFT          [4]         [3,2,1]     θ        0                    0                        0
               | RIGHT-ARC(1)   [2,4]       [1]         (2>3)    0                    0                        0
               | RIGHT-ARC(2)   [1,2,4]     []          (1>3)    0                    0                        0
               | LEFT-ARC(1)    [3,4]       [1]         (3>2)    1                    0.25                     1.25
               | LEFT-ARC(2)    [2,3,4]     []          (3>1)    1                    0.5                      1.5
```

Figure 6.2: Generating more than three parse states from one state.

The following sections describe different types of constraint rules that can be extracted from a dependency treebank, where they can be integrated into our parser.

6.5.2 Extracting constraint rules from PATB

The main type of relations that are accounted for in dependency parsing are the parent-daughter relations between different words in a sentence. We use the POS tags and some other information when extracting dependency relations between words in sentences. The main reason for using POS tags instead of word forms is that POS tags are more general than word forms and hence they allow for some generalisations in the rules. We devote the following sections to describing different constraint rules extracted from a set of dependency trees. The following section corresponds to our third contribution of this study, which is C.3 in Section 1.4.

6.5.3 Parent daughter relations extraction

From the dependency tree shown in Figure 6.3, we can specify the parent-daughter relations between the words in the sentence. These relations are easily obtained during parser training. At each parse step, the relations between two items are recorded when the parser performs LEFT-ARC or RIGHT-ARC operations. The parsing steps for the sentence in (14) are presented in Figure 6.4.

(14)
خمسون الف سائح زاروا لبنان و سورية في أيلول الماضي *xmswn alf sA'i.h zAraW lbnAn w swryT fy 'ylwl almA.dy* "fifty thousand tourists visited Lebanon and Syria last September" [Habash and Roth, 2009c]

For example, from the dependency tree in Figure 6.3 we can identify that زاروا *zArwA* "visited"$_{V4}$ is the parent of سائح *sA'i.h* "tourists"$_{N3}$, و *w* "and"$_{C6}$, and في *fy* "in"$_{P8}$. We can also identify that سائح *sA'i.h* "tourists"$_{N3}$ is the parent of خمسون *xmswn* "fifty"$_{NN1}$ and الف *laf* "thousand"$_{NN3}$, and so on. Using the information available

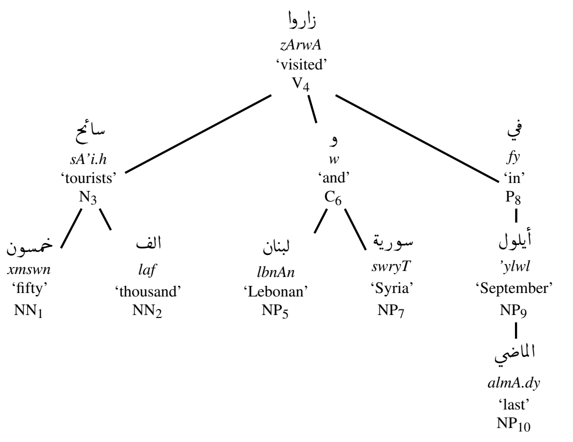


Figure 6.3: Dependency tree for the sentence in (14). The POS tags are abbreviated to accommodate space in Figure 6.4.

Dependency relations: (V4>N3) (V4>C6) (V4>P8) (N3>NN1) (N3>NN2) (C6>NP5) (C6>NP7) (P8>NP9) (NP9>NP10)

Steps	Action	Queue	Stack	RELS
1	θ	[NN1,NN2,N3,V4,NP5,C6,NP7,P8,NP9,NP10]	[]	θ
2	SHIFT	[NN2,N3,V4,NP5,C6,NP7,P8,NP9,NP10]	[NN1]	θ
3	SHIFT	[N3,V4,NP5,C6,NP7,P8,NP9,NP10]	[NN2,NN1]	θ
4	RIGHT-ARC	[N3,V4,NP5,C6,NP7,P8,NP9,NP10]	[NN1]	R1 = (N3>NN2)
5	RIGHT-ARC	[N3,V4,NP5,C6,NP7,P8,NP9,NP10]	[]	R2=R1∪(N3>NN1)
6	SHIFT	[V4,NP5,C6,NP7,P8,NP9,NP10]	[N3]	R2
7	RIGHT-ARC	[V4,NP5,C6,NP7,P8,NP9,NP10]	[]	R3=R2∪(V4>N3)
8	SHIFT	[NP5,C6,NP7,P8,NP9,NP10]	[V4]	R3
9	SHIFT	[C6,NP7,P8,NP9,NP10]	[NP5,V4]	R3
10	RIGHT-ARC	[C6,NP7,P8,NP9,NP10]	[V4]	R4=R3∪(C6>NP5)
11	SHIFT	[NP7,P8,NP9,NP10]	[C6,V4]	R4
12	LEFT-ARC	[C6,P8,NP9,NP10]	[V4]	R5=R4∪(C6>NP7)
13	LEFT-ARC	[V4,P8,NP9,NP10]	[]	R6=R5∪(V4>C6)
14	SHIFT	[P8,NP9,NP10]	[V4]	R6
15	SHIFT	[NP9,NP10]	[P8,V4]	R6
16	SHIFT	[NP10]	[NP9,P8,V4]	R6
17	LEFT-ARC	[NP9]	[P8,V4]	R7=R6∪(NP9>NP10)
18	LEFT-ARC	[P8]	[V4]	R8=R7∪(P8>NP9)
19	LEFT-ARC	[V4]	[]	R9=R8∪(V4>P8)
20	SHIFT	[]	[V4]	R9

Figure 6.4: Parse transitions for processing the sentence in (14) for extracting a set of relations between items.

in a dependency tree we parse each sentence using the tree as a grammar. When the parser performs LEFT-ARC or RIGHT-ARC operations, the relations between the parent and the daughter item are recorded. The dependency relations are then stored as a set of constraint rules for parsing test data, an example of a set of relations extracted from the parse transitions of Figure 6.4 is shown in Figure 6.5. The first line of Figure 6.4 says that سائح *sA'i.h* "tourists"$_{N3}$ is the parent of خمسون *xmswn* "fifty"$_{NN1}$ and so on.

```
(N3, NN1)
(N3, NN2)
(V4, N3)
(C6, NP5)
(C6, NP7)
(V4, C6)
(NP9, NP10)
(P8, NP9)
(V4, P8)
```

Figure 6.5: Extracted dependency relation between parents and daughter during parser training from Figure 6.4.

Each relation is recorded as a tuple where each tuple contains two items: the first item is the parent, and the second item is the daughter. A set of tuples is then used as constraint rules during parsing. When the parse model recommends a LEFT-ARC or a RIGHT-ARC operation, this could make `x` the parent of `y`. The parser can check from the set of constraint rules to see whether making `x` the parent of `y` is allowed or is plausible. A recommended operation that results in producing a dependency relation between two items that does not obey any constraint rules must not be encouraged; i.e., it must not be given a positive score. Intuitively, one can use information about the frequency of rules: A rule with high frequency rate should be given priority. However, previous experiments with using such information showed that they were not helpful. Hence, information about the frequency of rules is ignored. So, the computation of the scores is the sum of three scores:

1. the original score of the parse state.

2. the score for the recommendation of the parse model.

3. the score for obeying/disobeying a constraint rule.

We have evaluated the parser using the parent-daughter relations and the result for this evaluation is presented in Table 6.11.

UAS (%)	LAS (%)	LA (%)	Speed
70.1	66.7	93.4	0.059

Table 6.11: Evaluating DNDParser with parent-daughter constraint rules.

The output of using the kind of constraint rules that have been presented in this section is not encouraging. The results in Table 6.11 show that using the constraint rules slightly worsens the parsing performance. The reason for the worsening of the parsing performance is that the constraint rules are very generalised, hence many operations suggested by the parsing rules are encouraged by the constraint rules. For example, in Figure 6.4, in step 3 the correct operation is RIGHT-ARC for making `N3` the head of `NN2`. However, having a large number of overly generalised constraint rules may include a rule that allows `NN2` to be the head of `N3`. Introducing such a rule will encourage the parser to perform LEFT-ARC instead of RIGHT-ARC, which may not be the correct operation.

Although the application of constraint rules harms the parsing accuracy, they can actually improve the parsing speed substantially, from 0.081 seconds per relations to 0.059 seconds per relation. An explanation of this outcome is that the parser accepts many individual relations as plausible and hence produces a final parse analysis more quickly, even if it is not especially plausible.

It is likely that the major problem with the constraint rules above is they lack contextual information. In order to improve them we may include some contextual information, such as situations in which `x` appeared as the head of `y` and situations in which `y` appeared as the head of `x`. The extraction of this kind of constraint rule is discussed in Section 6.5.

6.5.4 Subtree extraction

From a set of dependency trees, we can extract a different set of constraint rules. In this section we describe a different set of constraint rules that we have extracted from the treebank. These constraint rules consist of different sets of subtrees. They include both lexicalised subtrees and unlexicalised subtrees. In lexicalised subtrees, each subtree is headed by a lexical item and has zero or more leaves. The leaves are stored in a list of tuples where the first element of each tuple is a number indicating the frequency of a set of leaves (which are a list POS tags) for the lexical item. These constraint rules are called lexicalised subtrees because each rule is headed by a lexical item, which is the head of the tree, and the list of items which is a set of POS tags for the daughters of the head. Figure 6.6 shows a set of lexicalised subtrees extracted from the dependency tree of Figure 6.3.

Lexicalised subtrees

```
‘zArwA’,    (1, [‘NOUN’, ‘CONJ’, ‘PREP’]).
‘sA’i.h’,   (1, [‘NOUN-NUM’, ‘NOUN-NUM’]).
‘w’,        (1, [‘NOUN-PROP’, ‘NOUN-NUM’]).
‘fy’,       (1, [‘NOUN-PROP’]).
‘’aylwl’,   (1, [‘NOUN-PROP’]).
‘xmswn’,    (1, []).
‘alf’,      (1, []).
‘lbnAn’,    (1, []).
‘swryT’,    (1, []).
‘almA.dy’,  (1, []).
```

Figure 6.6: A set of lexicalised subtrees extracted from Figure 6.3.

Since the LEFT-ARC and the RIGHT-ARC operations result in removing a daughter item from the stack or queue, which may be required in subsequent parsing stages, it is vital to ensure that the daughter item has collected all of its daughters. Removing an item from the stack or the queue when it is required in subsequent parsing steps will result in producing wrong trees, or possibly parse failure. The error propagation may be severe when LEFT-ARC or RIGHT-ARC operations are mistakenly performed. Thus, subtrees can be used for encouraging the parser to remove a daughter item only if there is evidence that the daughter item has collected all of and only its daughters (This corresponds to completeness and cohesion in Lexical Functional Grammar (LFG) [Kaplan and Bresnan, 1995]). This check is performed in two steps by using the subtrees: (i) collect all the daughters of the item from the tree that have been built by the parser, and (ii) find a subtree (from the set of subtrees collected during parser training) that is headed by the dependent item with the same set of daughters that are collected in (i). If a matching subtree is found then we can assume that the dependent item has all of its daughters and hence the parse operation can be encouraged.

Encouraging a parse operation using the subtrees is performed by giving a score to the parse operation. Each daughter in a subtree is associated with a score, which represents the frequency of the subtree during the training stage. The score is used for computing the percentage of the subtree with a specific set of daughters, which is computed by dividing the score associated with the daughter by the total associated scores of all other daughters headed by the same lexical item. The computed score is then used for encouraging the parse operation.

We have evaluated DNDParser using this set of constraint rules and the results of the evaluation presented in Table 6.12 indicate an improvement over the previous experiments, where DNDParser was evaluated without the application of the constraint rules. In the next section we will attempt to use a set of unlexicalised constraint rules.

Constraint rules	UAS (%)	LAS (%)	LA (%)	Speed
DNDParser with lexicalised subtrees	73.06	69.15	94.21	0.151

Table 6.12: Constraining DNDParser with a set of probabilistic lexicalised subtrees.

Unlexicalised subtrees The data in the set of lexicalised subtrees is sparse because it is possible that many of the lexical items that occur during testing have never been seen during the training. In order to overcome the problem of sparsity, we will extract a set of unlexicalised subtrees from the treebank. The set of unlexicalised subtrees are headed by a POS tag instead of a lexical item. The rules in Figure 6.7 show a set of unlexicalised subtrees extracted from the tree in Figure 6.3.

```
'VERB',      (1, ['NOUN','CONJ','PREP']).
'NOUN',      (1, ['NOUN-NUM', 'NOUN-NUM']).
'CONJ',      (1, ['NOUN-PROP', 'NOUN-NUM']).
'PREP',      (1, ['NOUN-PROP']).
'NOUN-PROP', (1, ['NOUN-PROP']).
'NOUN-PROP', (3, []).
'NOUN-NUM',  (2, []).
```

Figure 6.7: A set of unlexicalised subtrees extracted from Figure 6.3.

The result of the evaluation using a set of unlexicalised subtrees is shown in Table 6.13. We can see from this table that using unlexicalised subtrees improves the accuracy of DNDParser .

Constraint rules	UAS (%)	LAS (%)	LA (%)	Speed
DNDParser with unlexicalised subtrees	75.9	72.4	94.84	0.133

Table 6.13: Constraining DNDParser with a set of probabilistic unlexicalised subtrees.

6.5.5 Parent daughter relations extraction with local contextual information

Using a set of daughter-parent relations can be highly generalised and it may allow for establishing relations between different items that should not be allowed in some situations. In order to reduce the generalisation of the constraint rules obtained in the previous section, we may include a specification of the items that appear around or in between parents and the daughters in the rules.

For example, during parser training we use the dependency tree for each sentence as a grammar for parsing a given sentence. As in the previous section, during each LEFT-ARC or RIGHT-ARC transition, the relation between the parent and the daughter is recorded. Additionally, some items from the queue and the stack are collected and recorded with each relation. By using the parse transitions from Figure 6.4 we collect up to two items from the queue and two items from the stack for each relation that is established between words items[2]. Additionally, the frequency of the rule is counted and recorded in each relation where they are used for computing the probability of the rule during parsing. The conditional probability of each rule is computed in three steps: (i) obtaining the frequency of a rule, (ii) obtaining the sum of the frequency of all the rules with the same parent and daughter relations (regardless of the local information collected from the queue and the stack), (iii) dividing the number obtained in step (i) by the number obtained in step (ii). The probability of each rule is then used as a score for encouraging a parse operation suggested by the parsing rules.

From the training data, we have extracted a set of relations with varying window size for the queue and the stack. In Figure 6.8 we show an example of constraint rules with a window size of two items from the queue and two items from the stack, which are extracted from the parse transitions presented in Figure 6.4. The first line of the rules in Figure 6.8 says that سائح *sA'i.h* "tourists"$_{N3}$ is the parent of خمسون *xmswn* "fifty"$_{NN1}$, the two items on the queue are سائح *sA'i.h* "tourists"$_{N3}$ and زارو *zAraw* "visited"$_{V4}$ and the first item on the stack is خمسون *xmswn* "fifty"$_{NN1}$ while the '-' says that there is no second item on the stack. The last element of the rule is its frequency in the training data.

We have conducted many experiments using this type of constraint rule. We have used various window sizes for the queue and stack items for each constraint rules. The queue size and the stack size are set with a window size between two items and five

[2]The number of items collected from the queue and the stack may vary between *1 . . . n*.

```
(N3, NN1, [N3, V4, NN1, '-'], 1).
(N3, NN2, [N3, V4, NN2, NN1], 1).
(V4, N3, [V4, NP5, N3, '-'], 1).
(C6, NP5, [C6, NP7, NP5, V4], 1).
(C6, NP7, [NP7, P8, C6, V4], 1).
(V4, C6, [C6, P8, V4, '-'], 1).
(NP9, NP10, [NP10, '-', NP9, P8], 1).
(P8, NP9, [NP9, '-',  P8, V4], 1).
(V4, P8, [P8, '-', V4, '-'], 1).
```

Figure 6.8: Dependency relations with local information from the tree in Figure 6.3. Note: excluding the parent and the daughter from the local information made no difference to parsing performance.

items. The experiments presented in Table 6.14 shows the results for using various window sizes for the queue and the stack[3]. The evaluation results presented in Table 6.14 are based on 1-fold validation because our goal is to find out the window size of queue and stack that produce the best parsing performance.

Queue items	Stack items	UAS (%)	LAS (%)	LA (%)	Speed
2	2	75.3	70.2	93.08	0.098
2	3	75.1	69.7	93.15	0.104
2	4	74.1	69.9	93.06	0.099
2	5	73.9	68.9	93.0	0.105
3	2	76.4	71.2	93.05	0.097
3	3	76.6	71.3	93.07	0.102
3	4	76.1	70.8	93.05	0.104
3	5	76.0	70.8	93.05	0.108
4	2	77.3	72.0	93.01	0.101
4	**3**	**77.3**	**72.1**	**93.02**	**0.103**
4	4	77.3	72.0	93.03	0.109
4	5	77.3	72.0	93.03	0.111
5	2	77.1	71.8	92.99	0.108
5	3	77.2	71.9	93.01	0.112
5	4	77.0	71.7	93.02	0.111
5	5	76.8	71.6	92.98	0.118

Table 6.14: 1-fold validation when constraining DNDParser with a set of probabilistic parent-daughter relations with different window sizes for the queue and the stack.

We can identify that the best parsing performance is achieved on a 1-fold validation with a window size of four items on the queue and three items on the stack. In Table

[3]We have evaluated the parser using constraint rules with up to 5 items on the queue and 5 items on the stack because using a combination of more than 10 items yielded no improvement in parsing performance.

6.15, we have presented the performance of DNDParser using a 5-fold cross validations with the settings that produced the best parsing performance during the 1-fold validation, where the queue size is four items and the stack size is three items.

Queue items	Stack items	UAS (%)	LAS (%)	LA (%)	Speed
4	3	76.2	72.7	94.85	0.145

Table 6.15: 5-fold validation when constraining the data-driven parser with a set of probabilistic parent-daughter relations with four items in the queue and three items on the stack.

6.5.6 Combining constraints

Since the effect of each type of constraint rule on parsing performance is different, we would like to see their effect when they are combined. In this section we demonstrate the parsing performance involved when combining the set of dependency relations with the set of subtrees. In this evaluation process, the parser checks for a relation between two items from the set of relations collected during training stage and it encourages the parse operation if it finds a relation by giving the score outlined in Section 6.5.5 to the parse state otherwise it awards zero score. A similar process is performed in applying the unlexicalised subtree rules. If a subtree is found for the dependent item, then the score outlined in Section 6.5.4 is given. The result of this evaluation is presented in Table 6.16. Combining different constraints does not yield better results than using each constraint individually. However, using a combination of different constraint rules leads to better parsing accuracy than using none but the parsing speed degrades by about 36%.

Constraint rules	UAS	LAS	LA	Speed
Parent-daughter relations and unlexicalised subtrees	75.3%	71.8%	94.82%	0.127

Table 6.16: 5-fold cross validation of DNDParser with combined set of constraint rules.

6.6 A comparison with the state-of-the-art parser

In this section we compare the two different versions of our parser (DNDParser and constraint DNDParser (CDNDParser)) with the best result we have obtained for MaltParser, as shown in Table 5.3.

The results we have obtained previously for MaltParser, which is one of the state-of-the-art parsers, and the best performance for our parsers, which we have presented in the previous sections, are presented in Table 6.17. We can note that our purely data-driven parser (DNDParser) is 43% more efficient than MaltParser. Although the unlabelled attachment accuracy of our parser is slightly lower than that of MaltParser (0.7%), the labelled attachment score and the labelled accuracy is more accurate than MaltParser by 1% and 1.4% respectively. We anticipate that this improved accuracy of labelled attachment scores and labelled score is because we have used a different approach to MaltParser (see Section 5.4.1 for more details).

The use of constraint rules has improved the parsing accuracy of DNDParser but it has noticeably degraded its speed. This indicates that the use of constraint rules improves parsing accuracy at the expense of its speed. However, the use of constraint rules improved the parsing accuracy over the accuracy of MaltParser by 1% for unlabelled attachment score, 2.7% for labelled attachment score and 2.65% for labelled accuracy, while the parser remained as efficient as MaltParser.

Parser	UAS (%)	LAS (%)	LA (%)	Speed
MaltParser	75.2	70.0	92.2	0.144
DNDParser	74.5	71.0	93.6	0.081
CDNDParser	76.2	72.7	94.85	0.145

Table 6.17: Parse accuracies of MaltParser and the different version of DNDParser.

From the experiments that we have conducted in this chapter, we can note that applying constraints to a data-driven parser improves its accuracy but that the speed may degrade. The method that we have presented in this chapter where we use a scoring technique for activating/deactivating the application of constraints (see Section 6.5) allows for trading off between accuracy and speed.

6.7 Error analysis

From the CoNLL evaluation script we investigated the types of errors that the parser makes. In Figure 6.9 the parsing errors regarding the assignment of heads and dependents to items are shown. Two of the most frequently occurring items when the wrong head or the wrong dependent was attached to them were `PREP` and `CONJ`. The `PREP` occurred 4,279 times while the `CONJ` item occurred 1038 time. The parser attached the wrong head to `PREP` 43% of the time. The errors with `PREP` were related to the PP-attachment problem, which is hard to solve using a training data of small size, which in our experiments is 112,885 words. In contrast, the errors with the `CONJ` occurred because we have made the conjunctions the head of conjunction phrases and when we have converted PATB to dependency format we have changed the sentence-initial `CONJ` to *ICONJ*. Thus, the parser failed to identify the right heads for `CONJ` items. In experiments conducted by Alabbas and Ramsay [Alabbas and Ramsay, 2013] where they had made sentences in `CONJ` final during the PATB conversion to dependency format, the error rates with `CONJ` were lower. However, our experiments on MaltParser using the HPT V.4 (where coordinated items had lower priority in the head percolation table), did not yield higher results than HPT V.5 (where coordinated items had higher priority).

```
The overall error rate and its distribution over CPOSTAGs

--------------+-------+-------+------+-------+------+-------+-------
Error         | words | head  |   %  | dep   |   %  | both  |   %
Rate          |       | err   |      | err   |      | wrong |
--------------+-------+-------+------+-------+------+-------+-------
total         | 26834 |  6081 |  23% |  1872 |   7% |   462 |   2%
--------------+-------+-------+------+-------+------+-------+-------
NOUN          |  5957 |  1105 |  19% |   767 |  13% |   242 |   4%
PREP          |  4279 |  1857 |  43% |    16 |   0% |    15 |   0%
DET+NOUN      |  3060 |   500 |  16% |   272 |   9% |    56 |   2%
NOUN_PROP     |  2513 |   349 |  14% |   202 |   8% |    35 |   1%
DET+ADJ       |  1942 |   231 |  12% |   163 |   8% |    15 |   1%
PV            |  1585 |   425 |  27% |    20 |   1% |    19 |   1%
CONJ          |  1038 |   518 |  50% |     8 |   1% |     8 |   1%
IV            |   976 |   313 |  32% |    20 |   2% |    20 |   2%
ADJ           |   905 |   208 |  23% |    89 |  10% |     8 |   1%
SUB_CONJ      |   760 |   138 |  18% |    38 |   5% |     2 |   0%
ICONJ         |   713 |     0 |   0% |     0 |   0% |     0 |   0%
NUM           |   634 |   109 |  17% |    28 |   4% |     4 |   1%
POSS_PRON     |   534 |    11 |   2% |    94 |  18% |     1 |   0%
PRON          |   421 |    55 |  13% |    57 |  14% |    22 |   5%
REL_PRON      |   371 |     8 |   2% |     0 |   0% |     0 |   0%
NEG_PART      |   206 |    99 |  48% |     2 |   1% |     2 |   1%
NOUN_NUM      |   196 |    22 |  11% |     8 |   4% |     0 |   0%
DEM_PRON      |   177 |    15 |   8% |    38 |  21% |     8 |   5%
ADV           |   155 |    50 |  32% |     2 |   1% |     1 |   1%
PVSUFF_DO     |    84 |     0 |   0% |    21 |  25% |     0 |   0%
ABBREV        |    81 |    17 |  21% |     0 |   0% |     0 |   0%
VERB_PART     |    68 |    14 |  21% |     0 |   0% |     0 |   0%
IVSUFF_DO     |    60 |     3 |   5% |    23 |  38% |     1 |   2%
VERB          |    35 |    16 |  46% |     1 |   3% |     1 |   3%
PART          |    25 |     6 |  24% |     3 |  12% |     2 |   8%
EXCEPT_PART   |    16 |     4 |  25% |     0 |   0% |     0 |   0%
FOCUS_PART    |    14 |     3 |  21% |     0 |   0% |     0 |   0%
DET           |    11 |     2 |  18% |     0 |   0% |     0 |   0%
NO_FUNC       |     9 |     0 |   0% |     0 |   0% |     0 |   0%
DET+NUM       |     3 |     0 |   0% |     0 |   0% |     0 |   0%
INTERROG_PART |     3 |     2 |  67% |     0 |   0% |     0 |   0%
CV            |     1 |     0 |   0% |     0 |   0% |     0 |   0%
LATIN         |     1 |     0 |   0% |     0 |   0% |     0 |   0%
RC_PART       |     1 |     1 | 100% |     0 |   0% |     0 |   0%
--------------+-------+-------+------+-------+------+-------+-------
```

Figure 6.9: A sample error report from the CoNLL evaluation script.

6.8 Summary

In this chapter we have evaluated several machine learning classifiers that could be used for generating a parse model. We have also conducted a large number of experiment on our parser, where we have ran it deterministically and non-deterministically. We have learned from our experiments that running the parser non-deterministically produces better parsing accuracy but the parsing speed degrades substantially.

Also, we have shown an approach for extracting different types of constraint rules from a dependency treebank. Our experiments have shown that applying different type of constraint rules to a non-deterministic data-driven parser affect parsing performance (in terms of speed and accuracy) in different ways. Some types of constraint rules affect the parsing speed while other types affect the parsing accuracy. We have demonstrated that it is possible to improve the parsing accuracy by using a set of constraint rules, these are presented in Section 6.5.

Chapter 7

Conclusion, Contributions, and Future Work

7.1 Conclusion

In this research we have re-implemented a variation of a state-of-the-art parser (MaltParser) in order to be able to non-deterministically process a set of natural language sentences. We have modified the arc-standard version of MaltParser, which is based on a shift-reduce parsing algorithm. In this algorithm, the parser processes items from a queue and a stack. A queue consists of some input strings while a stack consists of items from the queue that have been looked at by the parser. The original algorithm establishes dependency relations, if possible, between the head of the queue and the top item on the stack. We have identified a limitation in this algorithm in that it cannot generate trees for non-projective sentences[1]. We have modified the original algorithm by allowing our parser to establish dependency relations between the head of the queue and one or more items on the stack, which include items that are buried deep inside the stack. We have shown in Section 5.2.1 that with this modification the parser is able to reproduce the correct analysis for a non-projective sentence while the original algorithm could not. This is one of the contributions of this research.

The modified algorithm may generate a number of new parse states from a single state. Each parse state is given a score based on two different factors: (i) if the parsing rules suggest that the parse operation (SHIFT, LEFT-ARC, or RIGHT-ARC) should be performed, and (ii) if the constraint rules agree with the recommendation made by

[1]non-projective sentences are those with overlapping arcs between words as shown in Figure 5.11 in Section 5.2.1.

the parsing rules. The parsing states are then sorted by their scores and stored in an agenda, where the state on the top of the agenda is explored first.

We have shown in Section 6.5.2 an approach for extracting different kinds of constraint rules from a dependency treebank. Our primary aim in re-implementing MaltParser was to identify a way of applying constraint rules to it and to investigate whether the application of constraint rules would affect the performance of the parser. We applied two different kinds of constraint rules to the parser. Our findings are encouraging for pursuing further research in this direction. We have learned that the application of constraint rules to data-driven parsing may improve parsing accuracy (1.2% for unlabelled attachment score, 1.7% for labelled attachment score, and 1.25% for labelled score). However, it can worsen the parsing speed by 44% (see chapter 5.2 for more details of the re-implementation of the original algorithm and the results of our experiments). The improvement of parsing accuracy using constraint rules is our second contribution. Also, with our implementation we can deactivate the application of constraint rules (by assigning a score of 0 to it) if we aim for speed rather than accuracy. Alternatively, we can activate these rules (by assigning a score of 1) if we aim for accuracy rather than speed. The ability to activate/deactivate the application of constraint rules is a further contribution.

The focus of the background chapters of this research was to investigate and highlight features of a number of parsing algorithms (top-down parsing, bottom-up parsing, left-corner parsing, shift-reduce parsing, and chart parsing).

Additionally, another background chapter was dedicated to investigating different frameworks for processing the syntax of natural languages. These frameworks were dependency frameworks and phrase structure frameworks. Also, we have briefly described the probabilistic parsing approaches.

Since we conducted all of the experiments presented in this thesis on Arabic, because we had access to the Penn Arabic Treebank, we have devoted Chapter 4 to briefly describing some of the structural complexities of Arabic that may pose challenges to parsers.

7.2 Contributions of the thesis

In this study we have made a number of contributions to the field:

C.1 We have presented an approach to integrating a set of constraint rules into a data-driven parser. In Section 5.2.2, we have discussed our approach for assigning scores to parse states and outlined one of the benefits of this approach, which is the ability to assign scores obtained from constraint rules to parse states. In Section 6.5.1 we have described the way we apply constraint rules to our parser, and in Sections 6.5.3, 6.5.4, 6.5.5 and 6.5.6 we have shown the effect of constraint rules on parsing performance.

C.2 We have demonstrated a technique for easily activating/deactivating the application of a set of constraint rules to data-driven parsing, which provides an option for trading off between accuracy and speed. See Section 6.5 for more details.

C.3 We have shown a number of ways for automatically and quickly extracting different types of constraint rules from a set of dependency trees. This approach has been demonstrated in Section 6.5.2.

C.4 We have modified the arc-standard algorithm of MaltParser to parse non-projective sentences. Our re-implementation of the original algorithm allows for parsing non-projective sentences. We have proved this technique by applying it to a well known non-projective Czeck sentence in Section 5.2.1.

Using our parser in data-driven mode is 43% more efficient than MaltParser and its labelled attachment score and the labelled accuracy is more accurate than MaltParser by 1% and 1.4% respectively. Applying a set of constraint rules to our parser improved its accuracy over the accuracy of MaltParser by 1% for the unlabelled attachment score, 2.7% for the labelled attachment score and 2.65% for labelled accuracy while the parser remained as efficient as MaltParser.

7.3 Future work

The main focus of this study has been on improving the performance of a data-driven parser by using a set of constraint rules, which are extracted from a set of dependency trees. For this purpose, we re-implemented a version of a state-of-the-art parser (the arc-standard algorithm of MaltParser). The modified version of the parser that is presented here has been tested on Arabic. Since it is possible to train the parser using a set of data from a treebank, and we can automatically extract constraint rules from a set of dependency trees, it is possible to test the parser on a different language where a treebank is available. One of the tasks for the future will be to obtain a treebank for a different language and train and test the parser on it in order to investigate its extendibility to other languages.

Additionally, since it is possible to integrate a set of constraint rules into the parser, which may act as grammatical rules, it may be possible to integrate a set of linguistically rich grammatical rules and run the parser as purely grammar-driven. More, it is possible to balance the weight of the parsing rules and that of the grammatical rules so that the parser can pay attention to both sources of information equally, which makes the parser completely hybrid. This would be another future task.

Some other authors such as Charniak [Charniak, 2000] used a re-ranking technique to selecting the best final tree analysis from a set of final trees that are produced by the parser. This technique led to some improvements to parsing accuracy. We would like to use a similar technique in the future. Since we give probability scores, which are obtained from constraint rules, we may use those scores to compute the scores for complete trees and select a tree with the highest score as the final parse tree.

References

[Aho and Ullman, 1972] Aho, A. V. and Ullman, J. D. (1972). *The Theory of Parsing, Translation, and Compiling*, volume 1. Prentice-Hall.

[Alabbas, 2013] Alabbas, M. (2013). *Textual Entailment for Arabic*. PhD Thesis, School of Computer Science, The Univeristy of Manchester.

[Alabbas and Ramsay, 2011] Alabbas, M. and Ramsay, A. (2011). Evaluation of dependency parsers for long Arabic sentences. In *Proceeding of International Conference on Semantic Technology and Information Retrieval (STAIR'11)*, pages 243–248, Putrajaya, Malaysia. IEEE.

[Alabbas and Ramsay, 2013] Alabbas, M. and Ramsay, A. M. (2013). Natural language inference for Arabic using extended tree edit distance with subtrees. *Journal of Artificial Intelligence Research*, 48:1–22.

[Attardi, 2006] Attardi, G. (2006). Experiments with a multilanguage non-projective dependency parser. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, CoNLL-X 2006, pages 166–170, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Attia, 2008] Attia, A. M. (2008). *Handling Arabic Morphological and Syntactic Ambiguities within the LFG Framework with a View to Machine Translation*. PhD Thesis, School of Languages, Linguistics and Cultures, Manchester University.

[Baptista, 1995] Baptista, M. (1995). On the nature of pro-drop in capeverdean creole. *Harvard Working Papers in Linguistics*, 5:3–17.

[Bauer, 2001] Bauer, M. (2001). A short introduction to x-bar syntax and transformations. *Language*, 77(4):863.

[Bengoetxea and Gojenola, 2010] Bengoetxea, K. and Gojenola, K. (2010). Application of different techniques to dependency parsing of Basque. In *Proceedings of the North American Chapter of the Association for Computational Linguistics Human Language Technologies 2010 First Workshop on Statistical Parsing of Morphologically-Rich Languages*, pages 31–39, USA. Association for Computational Linguistics.

[Bloomfield, 1933] Bloomfield, L. (1933). *Language*. The University of Chicago Press, Chicago.

[Bresnan and Kaplan, 1982] Bresnan, J. W. and Kaplan, R. (1982). Lexical function grammar. In Bresnan, J., editor, *The Mental Representation of Grammatical Relations*, pages 173–281, Cambridge, MA, USA. MIT Press.

[Briscoe and Carroll, 2002] Briscoe, T. and Carroll, J. (2002). Robust accurate statistical annotation of general text. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC-2002)*, Las Palmas, Canary Islands - Spain. European Language Resources Association (ELRA).

[Briscoe and Carroll, 2006] Briscoe, T. and Carroll, J. (2006). Evaluating the accuracy of an unlexicalized statistical parser on the PARC DepBank. In *Proceedings of the COLING/ACL on Main Conference Poster Sessions*, COLING-ACL 2006, pages 41–48, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Briscoe et al., 2006] Briscoe, T., Carroll, J., and Watson, R. (2006). The second release of the RASP system. In *Proceedings of the COLING/ACL on Interactive Presentation Sessions*, COLING-ACL 2006, pages 77–80, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Burnard, 2000] Burnard, L. (2000). *The British National Corpus Users Reference Guide*. Distributed by Oxford University Computing Services on behalf of the BNC Consortium.

[Chalabi, 2004a] Chalabi, A. (2004a). Elliptic personal pronoun and MT in Arabic. In *JEP-TALN 2004 session on Arabic Language Processing*, pages 189–191. Morocco.

[Chalabi, 2004b] Chalabi, A. (2004b). Sakhr Arabic lexicon. In *Proceedings of the NEMLAR International Conference on Arabic Language Resources and Tools*, pages 21–24, Cairo, Egypt.

[Chang and Lin, 2011] Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):1–27.

[Chapman, 1987] Chapman, D. (1987). Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377.

[Charniak, 1996] Charniak, E. (1996). Tree-bank grammars. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1031–1036, Menlo Park. AAAI Press/MIT Press.

[Charniak, 2000] Charniak, E. (2000). A maximum-entropy-inspired parser. In *Proceedings of the 1st North American Chapter of the Association for Computational Linguistics Conference*, NAACL 2000, pages 132–139, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Charniak et al., 1998] Charniak, E., Goldwater, S., and Johnson, M. (1998). Edge-based best-first chart parsing. In *Proceedings of the sixth workshop on very large corpora*, pages 127–133. Morgan Kaufmann.

[Chomsky, 1957] Chomsky, N. (1957). *Syntactic Structures*. Mouton, The Hague.

[Chomsky, 1959] Chomsky, N. (1959). On certain formal properties of grammars. *Information and Control*, 2(2):133–167.

[Chomsky, 1981] Chomsky, N. (1981). *Lectures on Government and Binding*. Foris Publications, Dordrecht.

[Cocke and Schwartz, 1970] Cocke, J. and Schwartz, J. T. (1970). Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University.

[Collins, 1997] Collins, M. (1997). Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics*, ACL '98, pages 16–23, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Collins, 1999] Collins, M. (1999). *Head-Driven Statistical Models for Natural Language Parsing*. PhD Thesis, Computer and Information Science, University of Pennsylvania.

[Collins, 2003] Collins, M. (2003). Head-driven statistical models for natural language parsing. *Computational Linguistics*, 29(4):589–637.

[Collins, 1996] Collins, M. J. (1996). A new statistical parser based on bigram lexical dependencies. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, ACL '96, pages 184–191, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Corazza and Satta, 2006] Corazza, A. and Satta, G. (2006). Cross-entropy and estimation of probabilistic context-free grammars. In *Proceedings of the main conference on Human Language Technology Conference of the North merican Chapter of the Association of Computational Linguistics*, HLT-NAACL '06, pages 335–342, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Crystal, 1980] Crystal, D. (1980). *A First Dictionary of Linguistics and Phonetics*. Deutsch, London.

[Daelemans et al., 2003] Daelemans, W., Zavrel, J., Van Der Sloot, K., and Van Den Bosch, A. (2003). TiMBL: Tilburg memory based learner, version 5.0, reference guide. *Research Group Technical Report Series*, (ILK 03-10).

[Daimi, 2001] Daimi, K. (2001). Identifying syntactic ambiguities in single-parse Arabic sentence. *Computers and the Humanities*, 35(3):333–349.

[Dalrymple et al., 1995] Dalrymple, M., Kaplan, R. M., Maxwell, J. T., and Zaenen, A., editors (1995). *Formal Issues in Lexical Functional Grammar*. Centre for Studies in Language and Information, Stanford, California.

[Debusmann, 2000] Debusmann, R. (2000). An introduction to dependency grammar. *Hausarbeit für das Hauptseminar Dependenzgrammatik SoSe*, 99:1–16.

[Diab, 2009] Diab, M. (2009). Second generation tools (AMIRA 2.0): Fast and robust tokenization, POS tagging, and base phrase chunking. In *Proceedings of the Second International Conference on Arabic Language Resources and Tools*, pages 285–288, Cairo.

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

[Edmonds, 1967] Edmonds, J. (1967). Optimum branchings*. *Journal of research of the National Bureau of Standards–B. Mathematics and Mathematical Physics*, 71B(4):233–240.

[Eisner, 1996] Eisner, J. M. (1996). Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th Conference on Computational Linguistics - Volume 1*, COLING '96, pages 340–345, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Farghaly and Shaalan, 2009] Farghaly, A. and Shaalan, K. (2009). Arabic natural language processing: Challenges and solutions. *ACM Computing Surveys*, 8(4):1–22.

[Fehri, 1993] Fehri, A. F. (1993). *Issues in the Structure of Arabic Clauses and Words*. Kluwer Academic Publishers, Dordrecht.

[Flickinger et al., 1985] Flickinger, D., Pollard, C., and Wasow, T. (1985). Structure-sharing in lexical representation. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, ACL 1985, pages 262–267, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Gazdar et al., 1985] Gazdar, G., Klein, E., Pullum, G. K., and Sag, I. (1985). *Generalised Phrase Structure Grammar*. Basil Blackwell, Oxford.

[Habash and Roth, 2009a] Habash, N. and Roth, R. M. (2009a). CATiB: The Columbia Arabic treebank. In *Proceedings of the Association for Computational Linguistics-International Joint Conference on Natural Language Processing 2009 Conference Short Papers*, ACLShort '09, pages 221–224, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Habash and Roth, 2009b] Habash, N. Rambow, O. and Roth, R. (2009b). MADA+TOKAN: a toolkit for Arabic tokenization, diacritization, morphological disambiguation, POS tagging, stemming and lemmatization. In *Proceedings of the 2nd International Conference on Arabic Language Resources and Tools (MEDAR)*, pages 102–109, Cairo, Egypt.

[Habash and Roth, 2009c] Habash, N. Faraj, R. and Roth, R. (2009c). Syntactic annotation in the Columbia Arabic treebank. In *Proceedings of the 2nd International Conference on Arabic Language Resources and Tools (MEDAR)*, pages 125–132, Cairo, Egypt.

[Han et al., 2009] Han, R., Donghong, J., Jing, W., and Mingyao, Z. (2009). Parsing syntactic and semantic dependencies for multiple languages with a pipeline approach. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning: Shared Task*, pages 97–102, USA. Association for Computational Linguistics.

[Harper and Helzerman, 1995] Harper, M. P. and Helzerman, R. A. (1995). Extensions to constraint dependency parsing for spoken language processing. *Computer Speech and Language*, 9:187–234.

[Hellwig, 1986] Hellwig, P. (1986). Dependency unification grammar. In *Proceedings of the 11th International Conference on Computational Linguistics (COLING-86)*, pages 195–198.

[Hellwig, 2003] Hellwig, P. (2003). Dependency unification grammar. In *Agel, V., Eichinger, L.M., Eroms, H.-W., Hellwig, P., Heringer, H. J. and Lobin, H. (eds)*, pages 593–635. Walter de Gruyter.

[Holes, 2004] Holes, C. (2004). *Arabic: Structures, Functions and Varieties*. Georgian University Press, Washington D.C.

[Hudson, 1976] Hudson, J. (1976). Walmadjari. In Dixon, R. M. W., editor, *Grammatical categories in Australian languages*, pages 653–666. Australian Institute of Aboriginal Studies, Canberra.

[Hudson, 1984] Hudson, R. (1984). *Word Grammar*. Basil Blackwell, Oxford.

[Jaf and Ramsay, 2013] Jaf, S. F. and Ramsay, A. (2013). Towards the development of a hybrid parser for natural languages. In Jones, A. V. and Ng, N., editors, *2013 Imperial College Computing Student Workshop*, volume 35 of *OpenAccess Series in Informatics (OASIcs)*, pages 49–56, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Johnasson et al., 1978] Johnasson, S., Leech, G. N., and Goodluck, H. (1978). *Manual of Information to Accompany the Lancaster-Oslo/Bergen Corpus of British English, for Use with Digital Computers*. Department of English, University of Oslo.

[Kaplan, 1973] Kaplan, R. M. (1973). *A General Syntactic Processor*. Algorithmics Press, New York.

[Kaplan and Bresnan, 1995] Kaplan, R. M. and Bresnan, J. (1995). *Lexical-Functional Grammar: A Formal System for Grammatical Representation*. MIT Press, Cambridge, MA, USA.

[Kasami, 1965] Kasami, T. (1965). An efficient recognition and syntax-analysis algorithm for context-free languages. Technical report, Air Force Cambridge Research Lab, Bedford, MA.

[Kay, 1973] Kay, M. (1973). *The MIND System*. Algorithmics Press, New York.

[Kepser, 2000] Kepser, S. (2000). A coalgebraic modelling of head-driven phrase structure grammar. Technical Report Technical Report of the SFB 441, University of Tubingen.

[King et al., 2003] King, T. H., Crouch, R., Riezler, S., Dalrymple, M., and Kaplan, R. M. (2003). The PARC 700 dependency bank. In *Proceedings of the 4th International Workshop on Linguistically Interpreted Corpora (LINC-03)*, pages 1–8.

[Klein and Manning, 2003] Klein, D. and Manning, C. D. (2003). Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics - Volume 1*, ACL 2003, pages 423–430, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Kornai and Pullum, 1990] Kornai, A. and Pullum, G. K. (1990). The x-bar theory of phrase structure. *Language*, 66(1):24–50.

[Kruijff, 2001] Kruijff, G.-J. M. (2001). *A Categorial-Modal Logical Architecture of Informativity: Dependency Grammar Logic and Information Structure*. PhD Thesis, Institute of Formal and Applied Linguistics, Charles University.

[Kübler et al., 2009] Kübler, S., McDonald, R., and Nivre, J. (2009). Dependency parsing. *Synthesis Lectures on Human Language Technologies*, 2(1):1–127.

[Kuhlmann and Nivre, 2010] Kuhlmann, M. and Nivre, J. (2010). Transition-based techniques for non-projective dependency parsing. *Northern European Journal of Language Technology*, 2(1):1–19.

[Lester, 1971] Lester, M. (1971). *Introductory Transformational Grammar of English*. Holt, Rineheart and Winston Inc.

[Liles, 1971] Liles, B. L. (1971). *An Introductory Transformational Grammar*. Prentice-Hall, Inc., New Jersey.

[Maamouri and Bies, 2004] Maamouri, M. and Bies, A. (2004). Developing an Arabic treebank: methods, guidelines, procedures, and tools. In *Proceedings of the Workshop on Computational Approaches to Arabic Script-based Languages*, pages 2–9, Geneva.

[MacDonald, 2006] MacDonald, R. (2006). *Discriminative Learning and Spanning Tree Algorithms for Dependency Parsing*. PhD Thesis, Computer and Information Science, the University of Pennsylvania.

[Magerman, 1995] Magerman, D. M. (1995). Statistical decision-tree models for parsing. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, ACL '95, pages 276–283, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Marcus et al., 1993] Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a large annotated corpus of English: The penn treebank. *Computational Linguistics*, 19(2):313–330.

[Marton et al., 2010] Marton, Y., Habash, N., and Rambow, O. (2010). Improving Arabic dependency parsing with lexical and inflectional morphological features. In *Proceedings of the North American Association for Computational Linguistics-Human Language Technologies 2010 First Workshop on Statistical Parsing of Morphologically-Rich Languages*, SPMRL '10, pages 13–21, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Marton et al., 2013] Marton, Y., Habash, N., and Rambow, O. (2013). Dependency parsing of modern standard Arabic with lexical and inflectional features. *Computational Linguistics*, 39(1):161–194.

[Mathews, 1998] Mathews, C. (1998). *An Introduction to Natural Language Processing Through Prolog*. Addison Wesley Longman Limited.

[McDonald et al., 2005] McDonald, R., Crammer, K., and Pereira, F. (2005). Online large-margin training of dependency parsers. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, pages 91–98, USA. Association for Computational Linguistics.

[McDonald et al., 2006] McDonald, R., Lerman, K., and Pereira, F. (2006). Multilingual dependency parsing with a two-stage discriminative parser. In *Tenth Conference on Computational Natural Language Learning (CoNLL-X)*, New York.

[Meixun et al., 2011] Meixun, J., Hwidong, N., and Jong-Hyeok, L. (2011). Beyond chart parsing: An analytic comparison of dependency chart parsing algorithms. In *Proceedings of the 12th International Conference on Parsing Technologies*, pages 220–224, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Melčuk, 1979] Melčuk, I. A. (1979). *Studies in Dependency Syntax*. Karoma Publishers, Inc.

[Melčuk, 1988] Melčuk, I. A. (1988). *Dependency Syntax: Theory and Practice*. State University of New York Press, Albany.

[Miyao and Tsujii, 2008] Miyao, Y. and Tsujii, J. (2008). Feature forest models for probabilistic HPSG parsing. *Computional Linguistics*, 34(1):35–80.

[Nelken and Shieber, 2005] Nelken, R. and Shieber, S. M. (2005). Arabic diacritization using weighted finite-state transducers. In *Proceedings of the Association for Computational Linguistics Workshop on Computational Approaches to Semitic Languages*, Semitic '05, pages 79–86, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Nivre, 2005] Nivre, J. (2005). Dependency grammar and dependency parsing. technical report MSI report 05133. Technical report, Växjö University.

[Nivre, 2006] Nivre, J. (2006). *Inductive dependency parsing*, volume 34 of *Text, Speech and Language Technology*. Springer.

[Nivre, 2008] Nivre, J. (2008). Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34:513–553.

[Nivre et al., 2006] Nivre, J., Hall, J., and Jens, N. (2006). MaltParser: A data-driven parser-generator for dependency parsing. In *Proceedings of the 5th International Conference on Language Resources and Evaluation*, LREC 2006, pages 2216–2219.

[Nivre et al., 2007] Nivre, J., Hall, J., Kubler, S., Mcdonald, R., Nilsson, J., Riedel, S., and Yuret, D. (2007). The CoNLL 2007 shared task on dependency parsing. In *Proceedings of CoNLL. Slav Petrov, Dipanjan*.

[Nivre et al., 2010] Nivre, J., Rimell, L., McDonald, R., and Gòmez-Rodrìguez, C. (2010). Evaluation of dependency parsers on unbounded dependencies. In *Proceedings of the 23rd International Conference on Computational Linguistics*, COLING '10, pages 833–841, Beijing.

[Norvig, 1991] Norvig, P. (1991). Techniques for automatic memoization with applications to context-free parsing. *Computional Linguistics*, 17(1):91–98.

[Øvrelid et al., 2009] Øvrelid, L., Kuhn, J., and Spreyer, K. (2009). Improving data-driven dependency parsing using large-scale LFG grammars. In *Proceedings of the Association for Computational Linguistics-International Joint Conference on Natural Language Processing 2009 Conference Short Papers*, pages 37–40, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Pareschi and Miller, 1990] Pareschi, R. and Miller, D. (1990). Extending definite clause grammars with scoping constructs. In *7th International Conference in Logic Programming*, pages 373–389, Cambridge, MA, USA. MIT Press.

[Pereira and Warren, 1983] Pereira, F. C. N. and Warren, D. H. D. (1983). Parsing as deduction. In *Proceedings of the 21st annual meeting of the Association for Computational Linguistics*, pages 137–144, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Pollard and Sag, 1994] Pollard, C. J. and Sag, I. A. (1994). *Head-driven Phrase Structure Grammar*. Chicago University Press, Chicago.

[Ramsay, 1980] Ramsay, A. M. (1980). Parsing English text. In *Artificial Intelligence and Simulation of Behaviour Conference*, Amsterdam.

[Ramsay and Mansour, 2006] Ramsay, A. M. and Mansour, H. (2006). Local constraints on Arabic word order. In *Advances in Natural Language Processing (Fifth International Conference on NLP, FinTAL 2006)*, volume 4139 of *Lecture Notes in Artificial Intelligence*, pages 447–457, Turku, Finland. Springer.

[Ratnaparkhi, 1999] Ratnaparkhi, A. (1999). Learning to parse natural language with maximum entropy models. *Machine Learning*, 34(1-3):151–175.

[Sag and Wasow, 1999] Sag, I. A. and Wasow, T. (1999). *Syntactic theory: a formal introduction*. CSLI, Stanford, Ca.

[Sampson, 2015] Sampson, G. (2015 (accessed July 14, 2015)). *The SUSANNE Corpus: Documentation*. `http://www.grsampson.net/SueDoc.html`.

[Schröder, 2002] Schröder, I. (2002). *Natural Language Parsing with Graded Constraints*. PhD Thesis, Department of Computer Science, Hamburg University.

[Sgall et al., 1986] Sgall, P., Hajiá, E., and Panevová, J. (1986). *The Meaning of the Sentence in Its Pragmatic Aspects*. Reidel.

[Smrž et al., 2008] Smrž, O., Bielický, V., Kourilová, I., Krácmar, J., Hajič, J., and Zemánek, P. (2008). Prague Arabic dependency treebank: a word on the million words. In *Proceedings of the Workshop on Arabic and Local Languages (LREC 2008)*, pages 16–23, Marrakech, Morocco.

[Smrž and Hajic, 2006] Smrž, O. and Hajic, J. (2006). *The Other Arabic Treebank: Prague Dependencies and Functions*. In Arabic Computational Linguistics: Current Implementations, CSLI Publications.

[Socher et al., 2013] Socher, R., Bauer, J., Manning, C. D., and Ng, A. Y. (2013). Parsing with compositional vector grammars. In *Proceedings of the Association for Computational Linguistics Conference*, pages 455–465, Sofia, Bulgaria.

[Tapanainen and Järvinen, 1997] Tapanainen, P. and Järvinen, T. (1997). A non-projective dependency parser. In *Proceedings of the Fifth Conference on Applied Natural Language Processing*, ANLC '97, pages 64–71, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Tayli and Al-Salamah, 1990] Tayli, M. and Al-Salamah, A. (1990). Building bilingual microcomputer systems. *Communications of the ACM*, 33(5):495–505.

[Tesnière, 1959] Tesnière, L. (1959). *Eléments de Syntaxe Structurale*. Editions Klincksieck.

[Thant et al., 2011] Thant, W. W., Htew, T. M., and Thein, N. L. (2011). Context free grammar based top-down parsing of myanmar sentences. In *International Conference on Computer Science and Information Technology*, pages 71–75. Pattaya.

[Vijay-Shanker and Joshi, 1988] Vijay-Shanker, K. and Joshi, A. K. (1988). Feature structures based tree adjoining grammars. In *Proceedings of the 12th Conference on Computational Linguistics - Volume 2*, COLING '88, pages 714–719, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Xia and Palmer, 2001] Xia, F. and Palmer, M. (2001). Converting dependency structures to phrase structures. In *Proceedings of the 1st Human Language Technology Conference (HLT-2001)*, pages 1–5, San Diego.

[Yamada and Matsumoto, 2003] Yamada, H. and Matsumoto, Y. (2003). Statistical dependency analysis using support vector machines. In *In Proceedings of the 8th International Workshop on Parsing Technologies*, pages 195–206, Nancy, France.

[Younger, 1967] Younger, D. H. (1967). Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208.

[Zhu et al., 2013] Zhu, M., Zhang, Y., Chen, W., Zhang, M., and Zhu, J. (2013). Fast and accurate shift-reduce constituent parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 434–443, Sofia, Bulgaria. Association for Computational Linguistics.

[Zitouni et al., 2006] Zitouni, I., Sorensen, J. S., and Sarikaya, R. (2006). Maximum entropy based restoration of arabic diacritics. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 577–584, Stroudsburg, PA, USA. Association for Computational Linguistics.

Appendix A

Parse Trees

A.1 Top-down parse trees

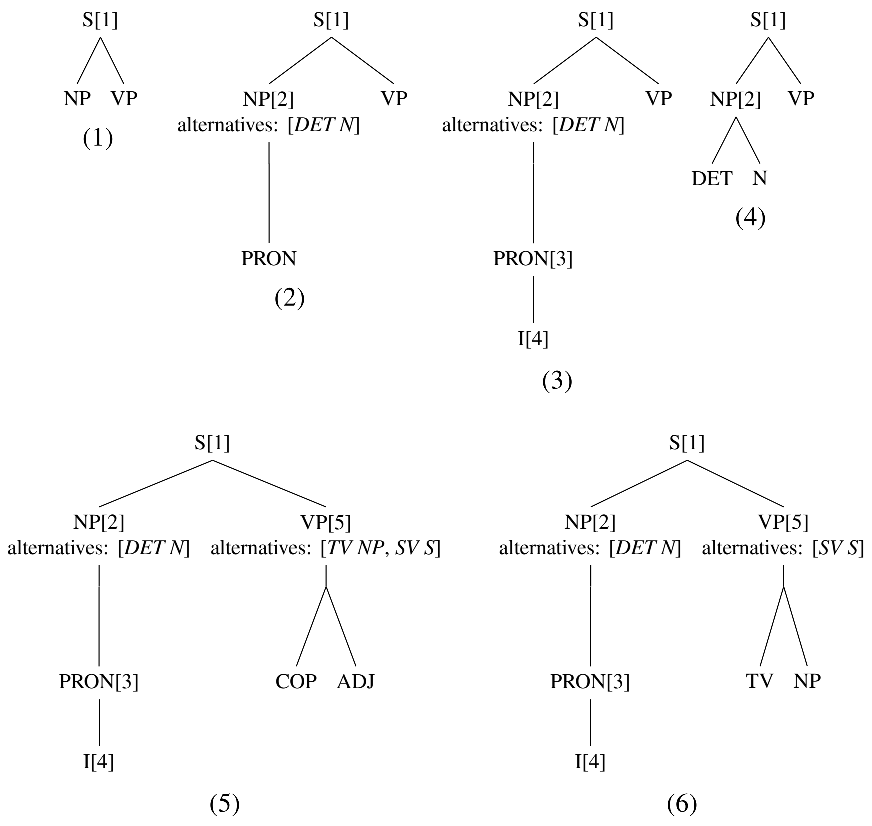
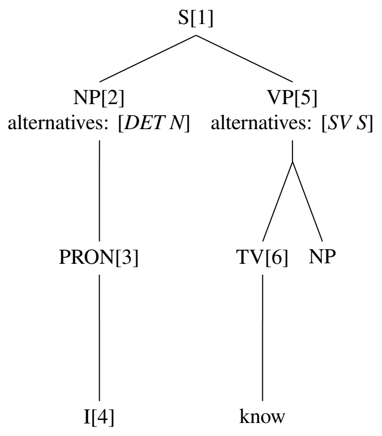
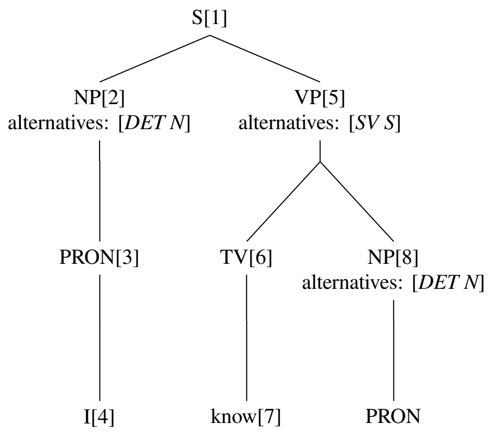


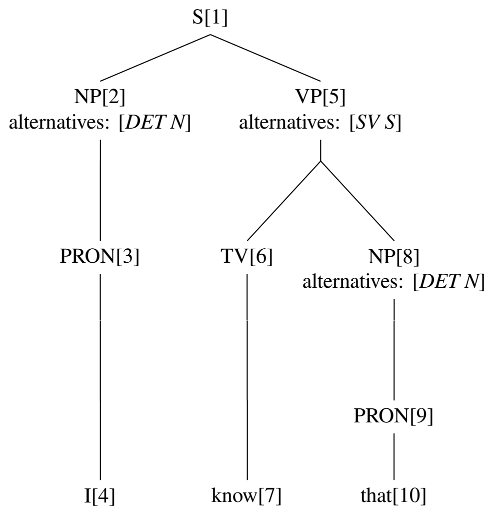
Figure A.1: Top-down parse trees, from tree (1) to tree (6).



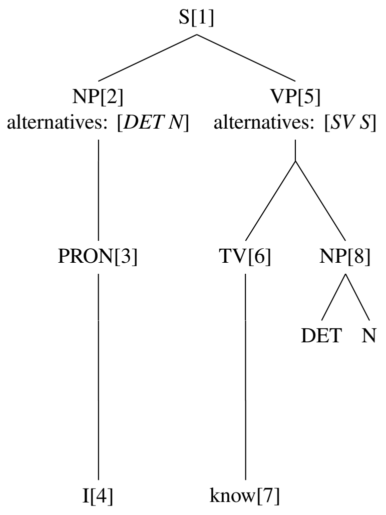
(7)



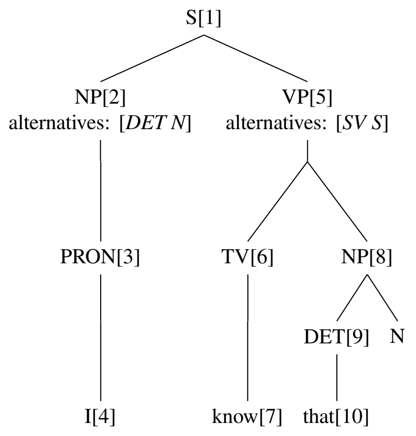
(8)



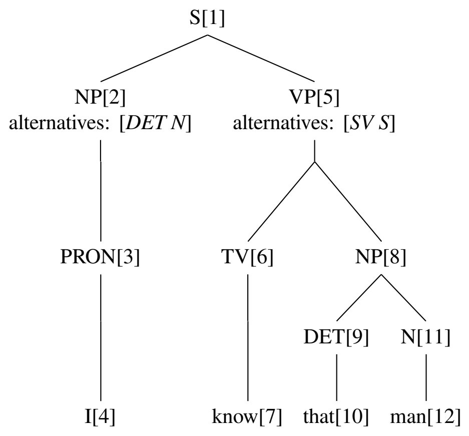
(9)



(10)



(11)



(12)

Figure A.1: Top-down parse trees, from tree (7) to tree (12).

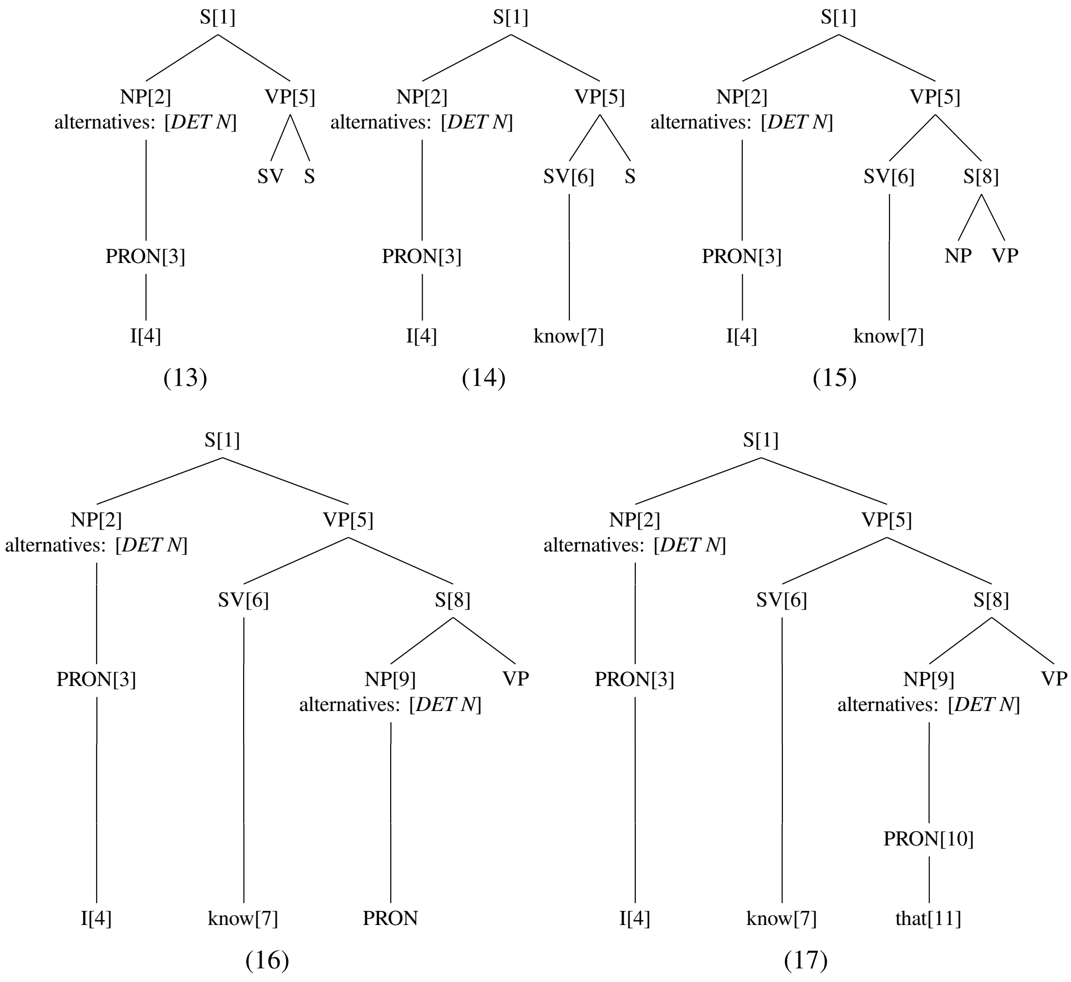
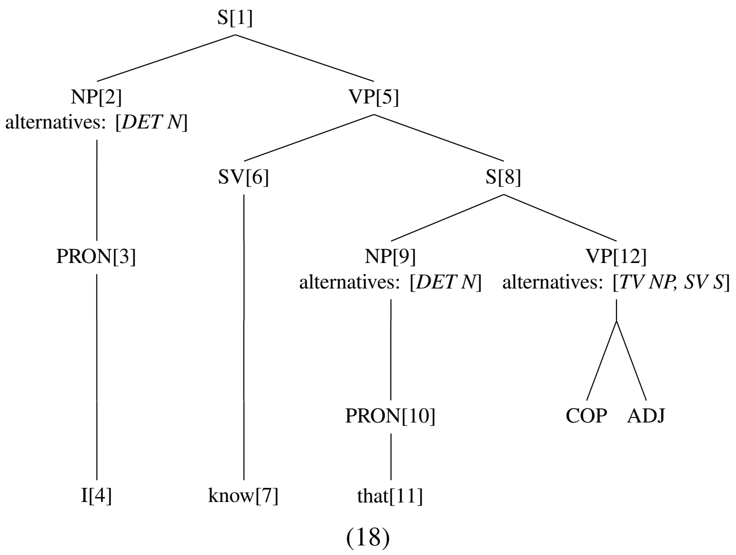
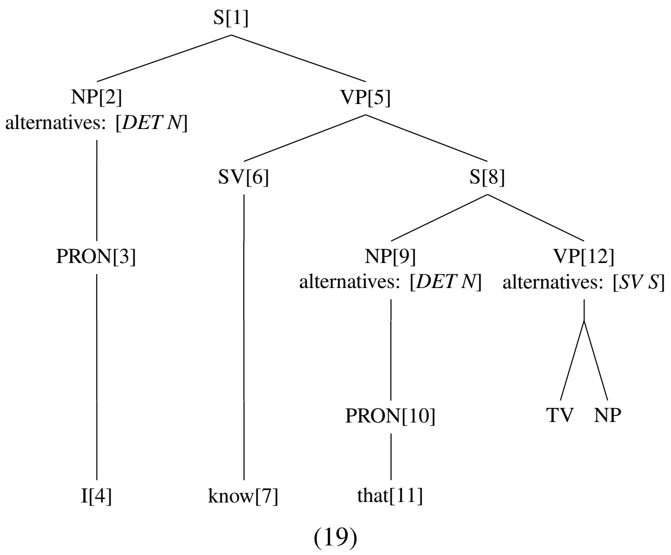


Figure A.1: Top-down parse trees, from tree (13) to tree (17).



(18)



(19)

Figure A.1: Top-down parse trees (18) and (19).

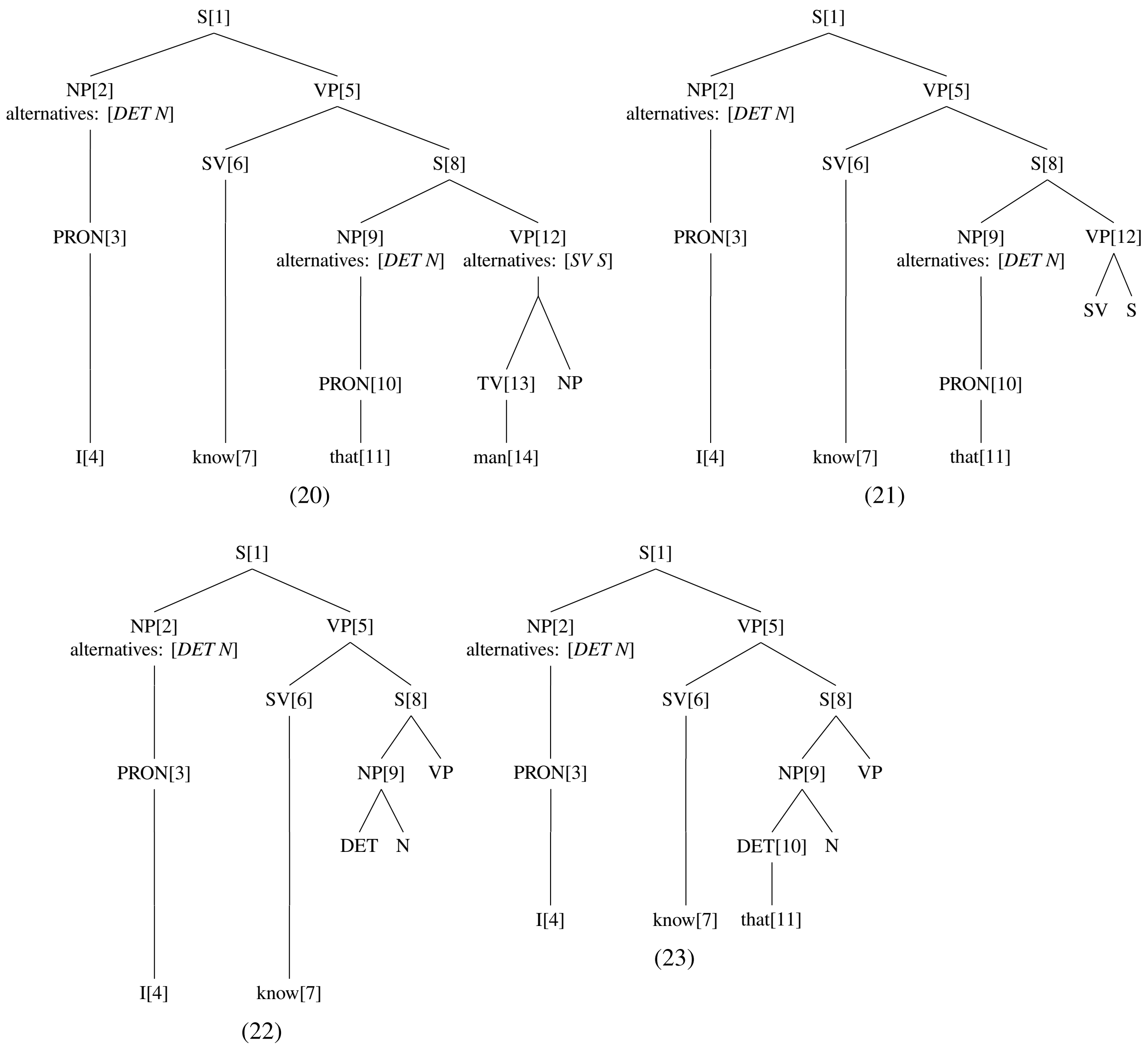
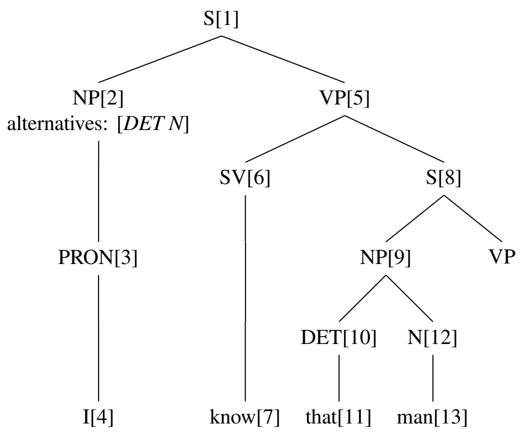
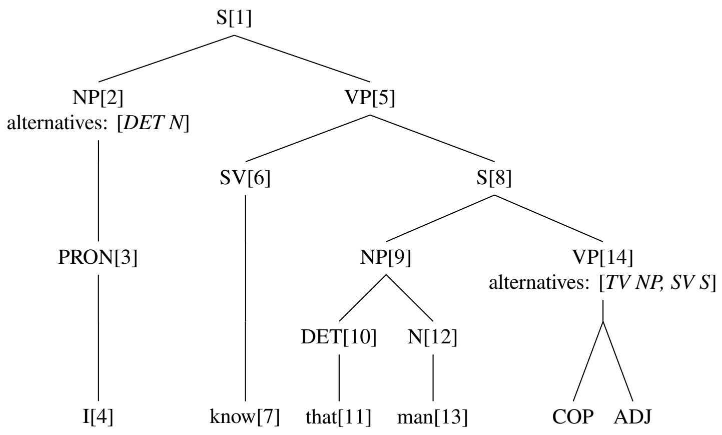


Figure A.1: Top-down parse trees, from tree (20) to tree (23).

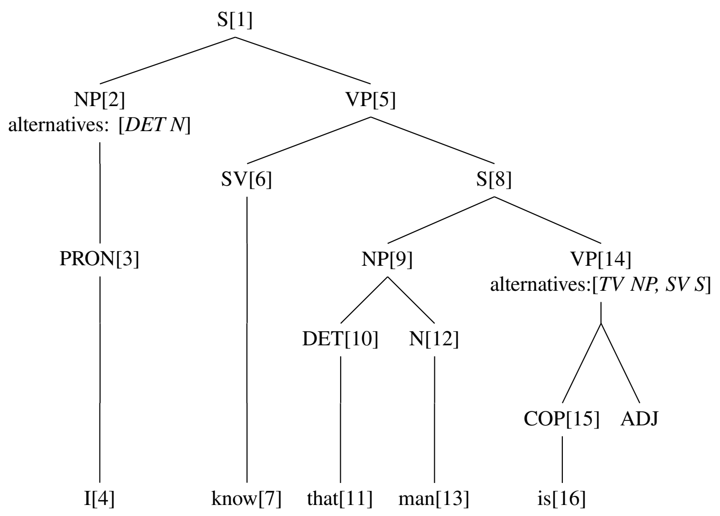


(24)

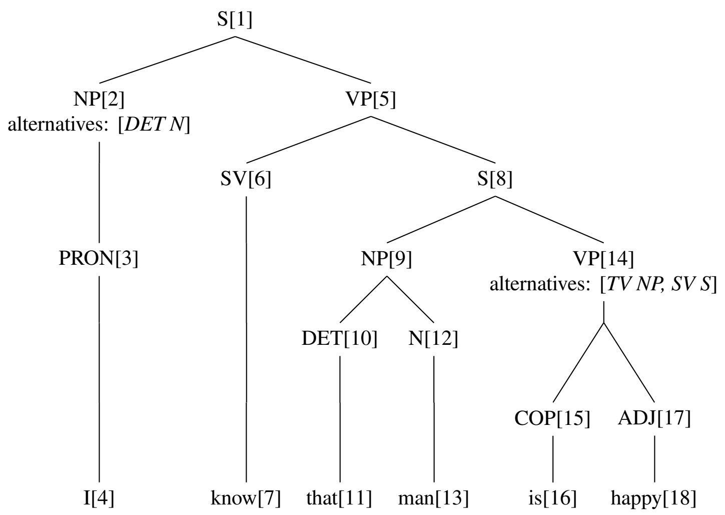


(25)

Figure A.1: Top-down parse trees (24) and (25).



(26)



(27)

Figure A.1: Top-down parse trees (26) and (27).

A.2 Bottom-up parsing

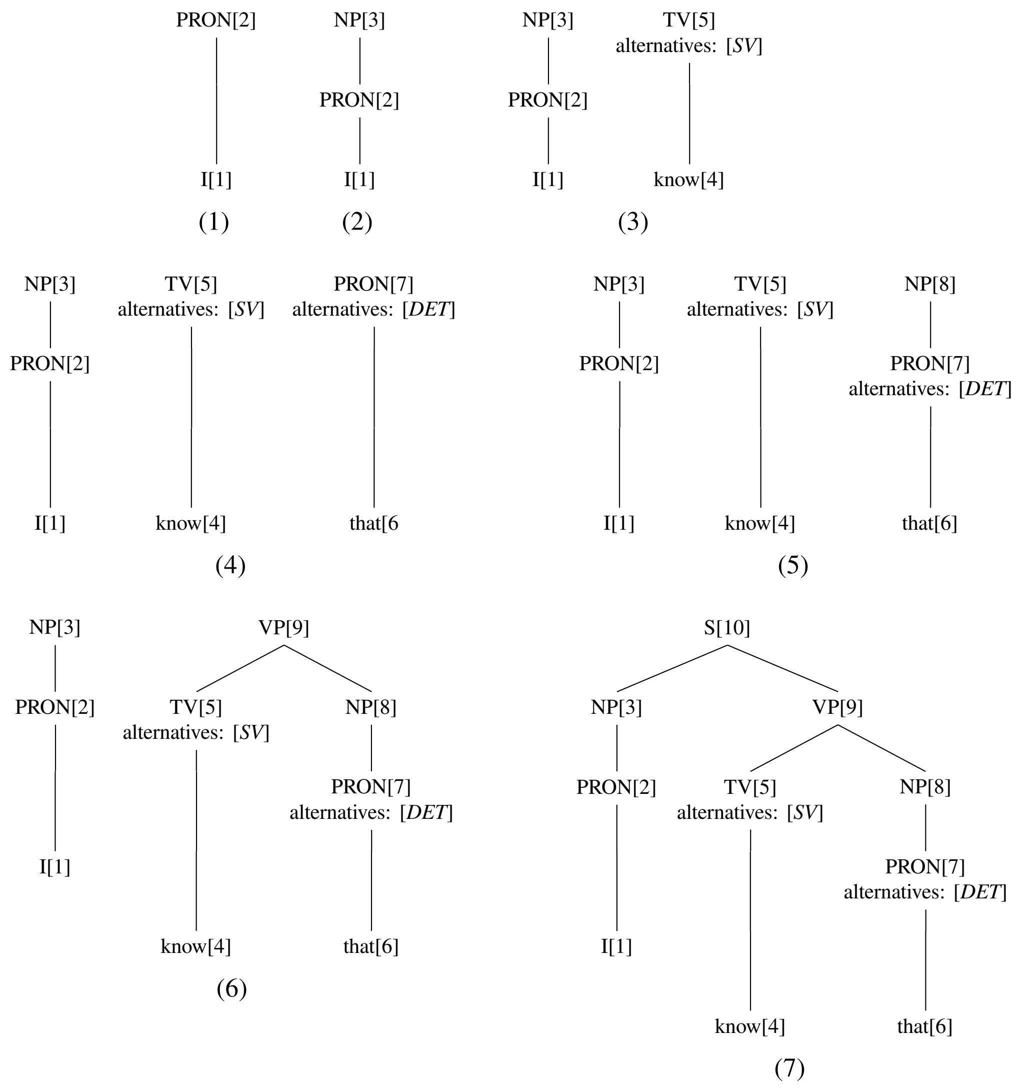


Figure A.2: Bottom-up parse trees, from tree (1) to tree (7).

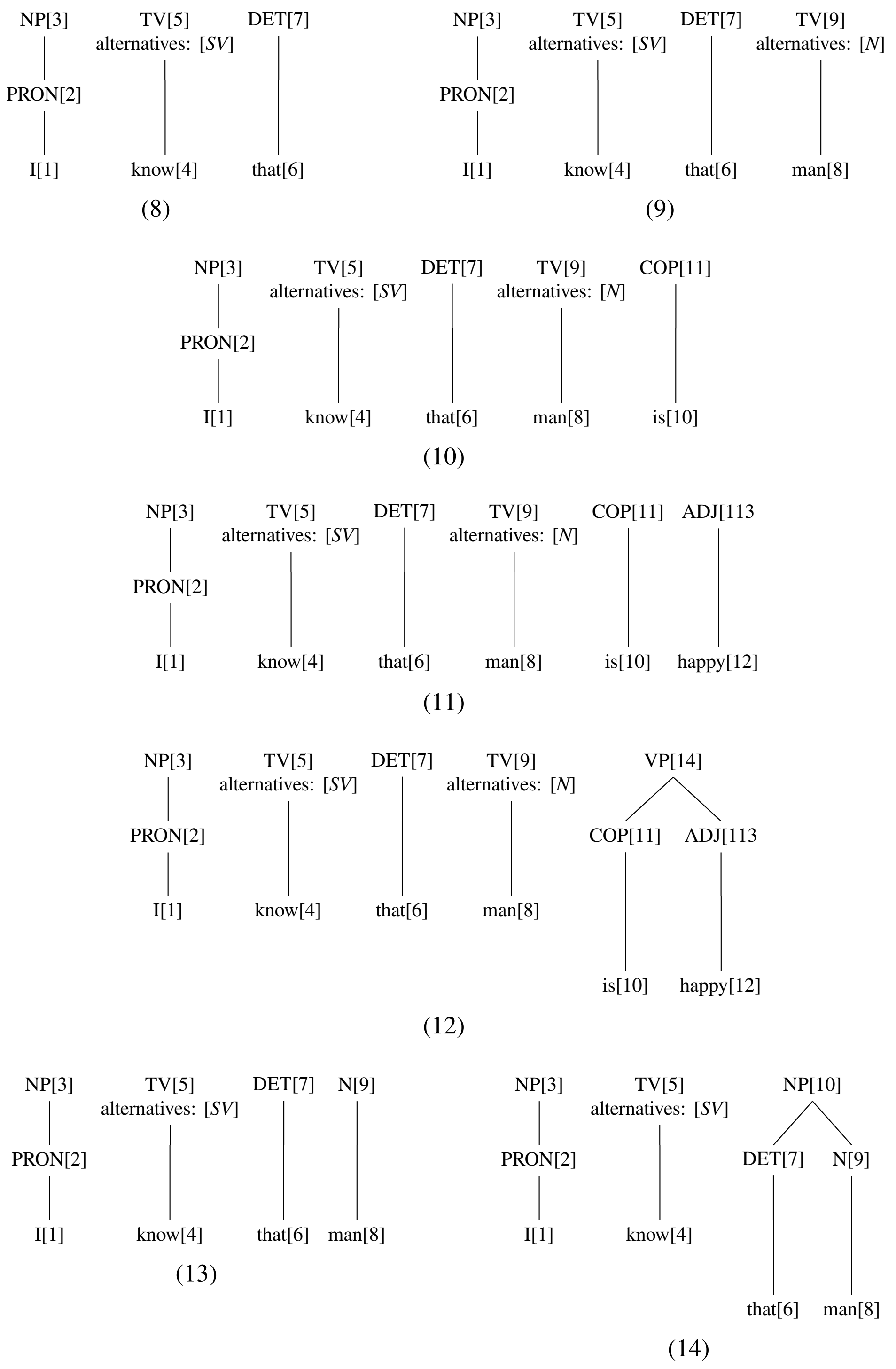


Figure A.2: Bottom-up parse trees, from tree (8) to tree (14).

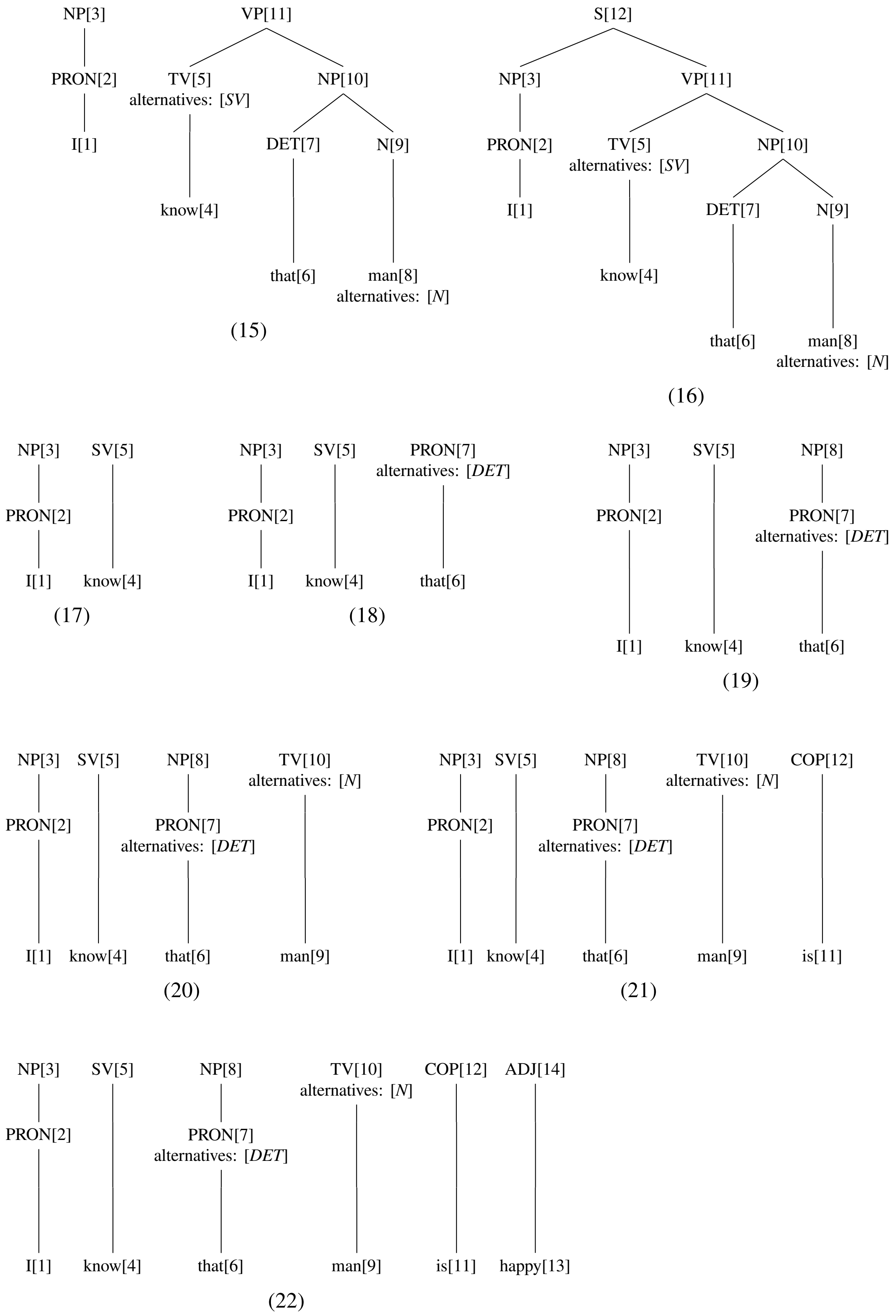


Figure A.2: Bottom-up parse trees, from tree (15) to tree (22).

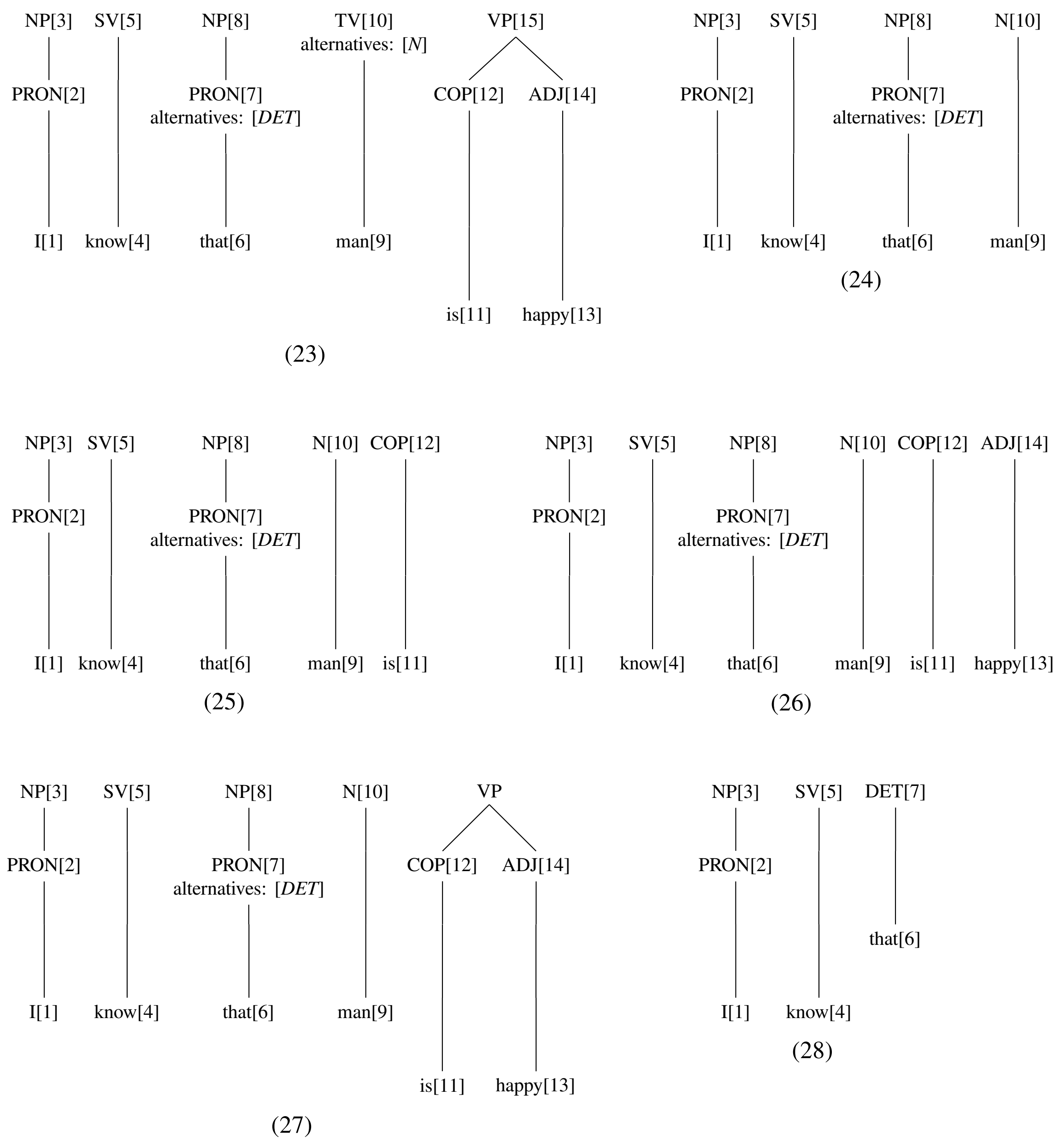


Figure A.2: Bottom-up parse trees, from tree (23) to tree (28).

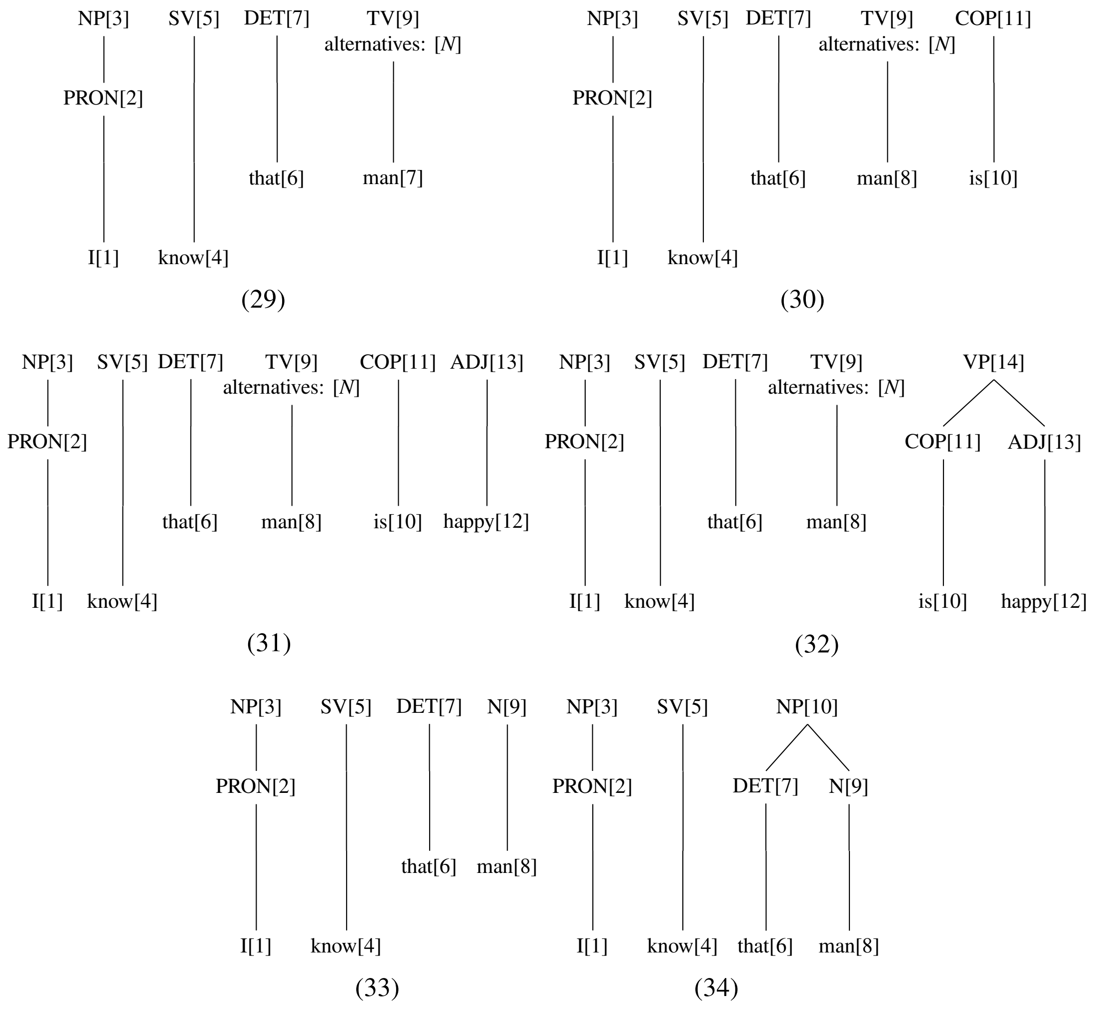


Figure A.2: Bottom-up parse trees, from tree (29) to tree (34).

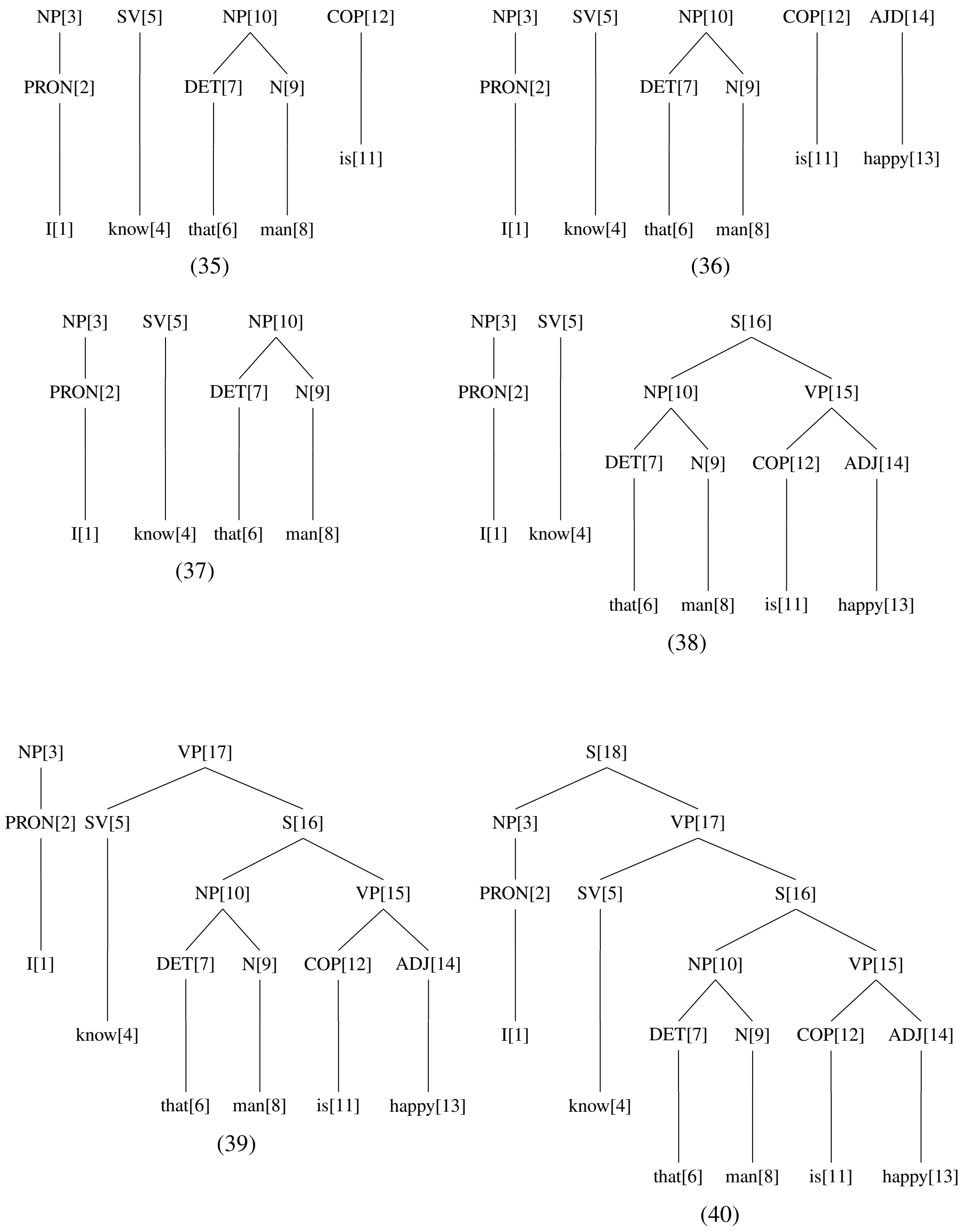

NP[3] SV[5] NP[10] COP[12]
PRON[2] DET[7] N[9]
is[11]
I[1] know[4] that[6] man[8]
(35)
NP[3] SV[5] NP[10] COP[12] AJD[14]
PRON[2] DET[7] N[9]
is[11] happy[13]
I[1] know[4] that[6] man[8]
(36)
NP[3] SV[5] NP[10]
PRON[2] DET[7] N[9]
I[1] know[4] that[6] man[8]
(37)
NP[3] SV[5] S[16]
PRON[2] NP[10] VP[15]
DET[7] N[9] COP[12] ADJ[14]
I[1] know[4]
that[6] man[8] is[11] happy[13]
(38)
NP[3] VP[17]
PRON[2] SV[5] S[16]
NP[10] VP[15]
I[1] DET[7] N[9] COP[12] ADJ[14]
know[4]
that[6] man[8] is[11] happy[13]
(39)
S[18]
NP[3] VP[17]
PRON[2] SV[5] S[16]
NP[10] VP[15]
I[1] DET[7] N[9] COP[12] ADJ[14]
know[4]
that[6] man[8] is[11] happy[13]
(40)

Figure A.2: Bottom-up parse trees, from tree (35) to tree (40)

A.3 Left-corner parse trees

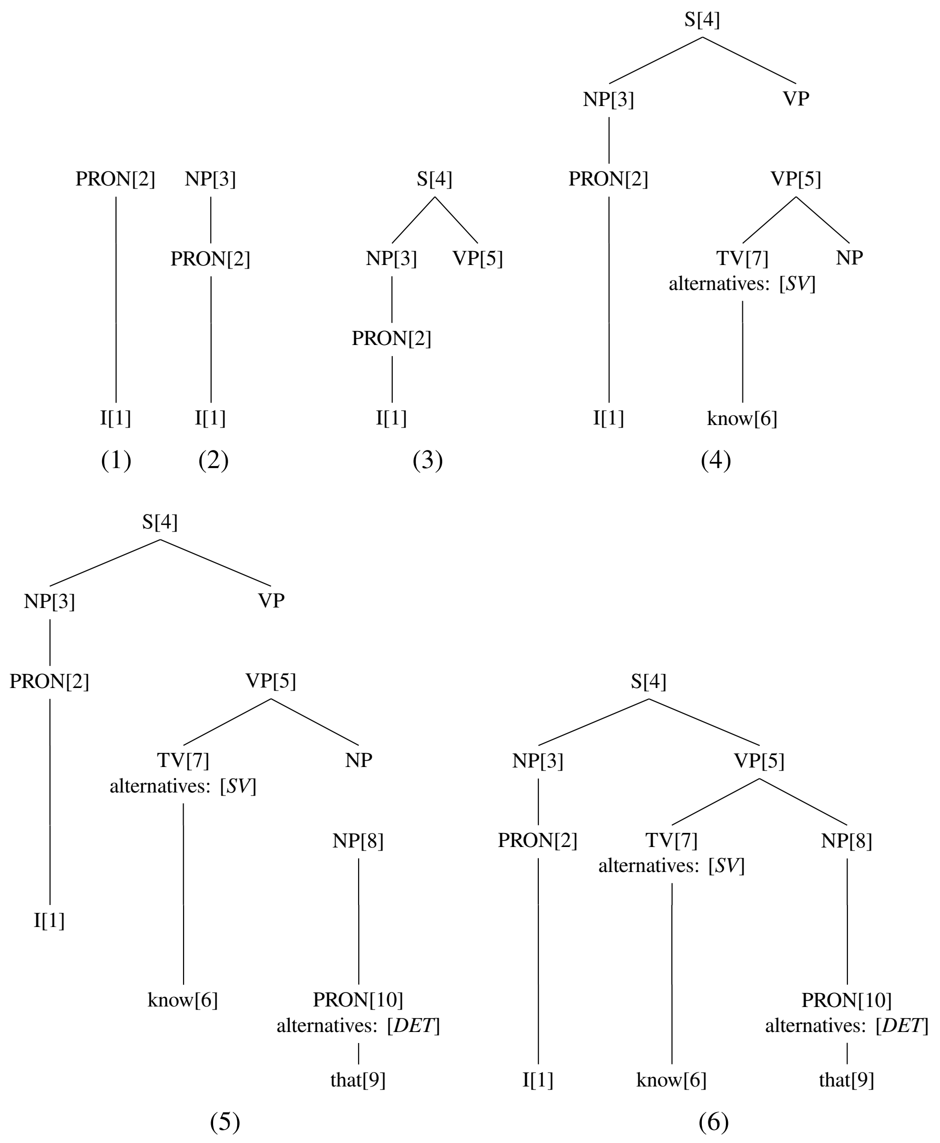


Figure A.3: Left-corder parse trees, from tree (1) to tree (6).

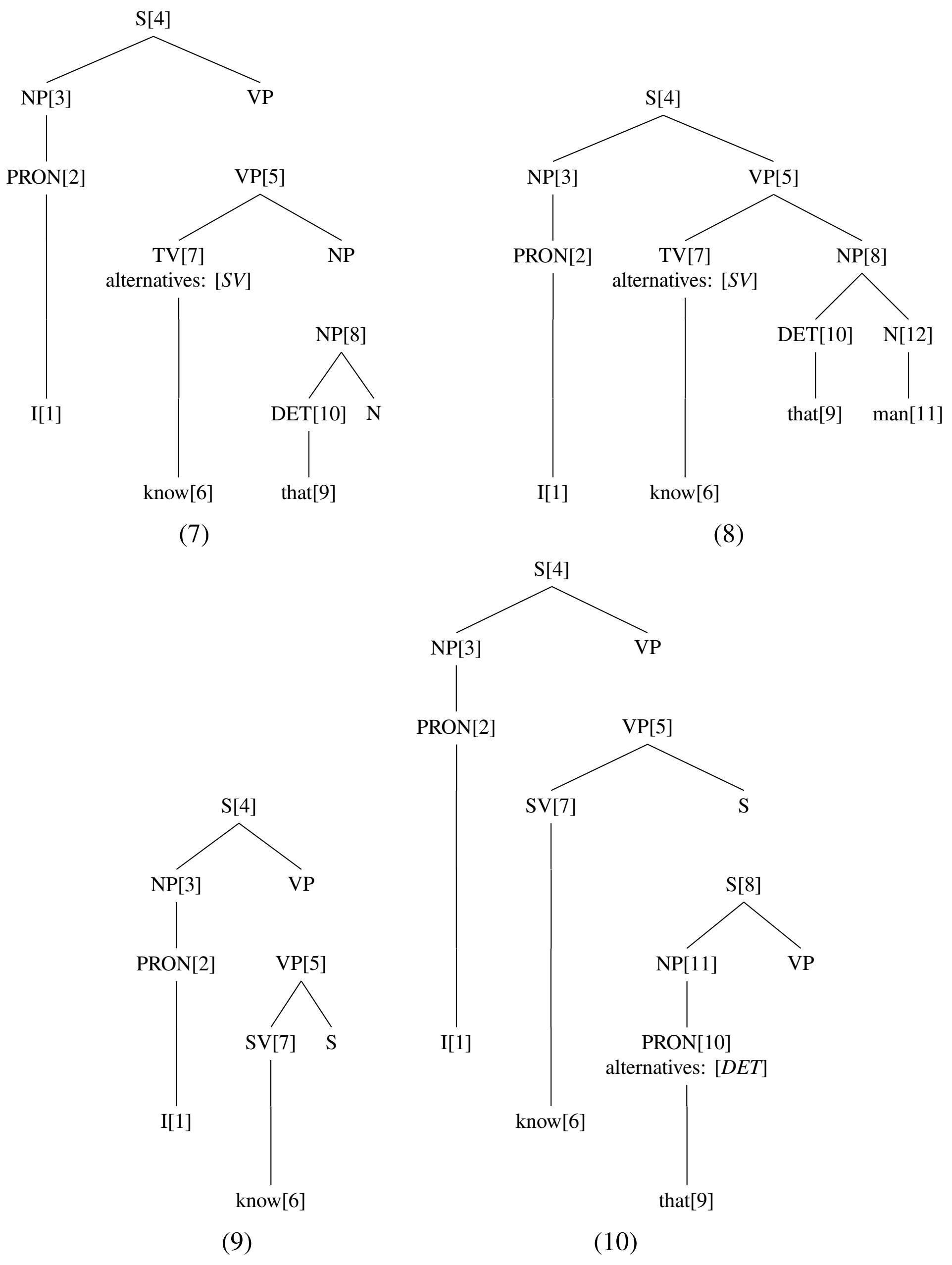


Figure A.3: Left-corder parse trees, from tree (7) to tree (10).

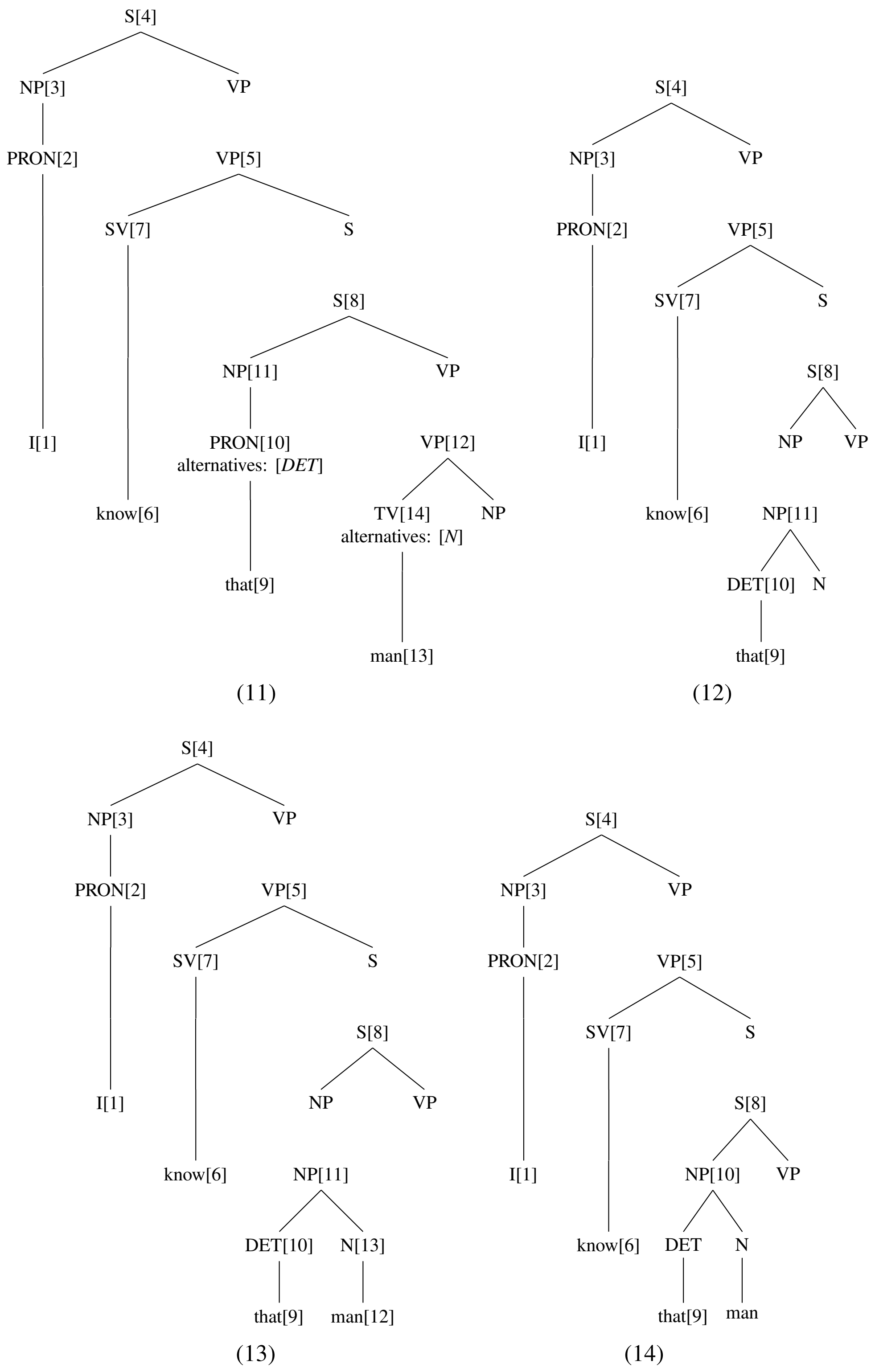
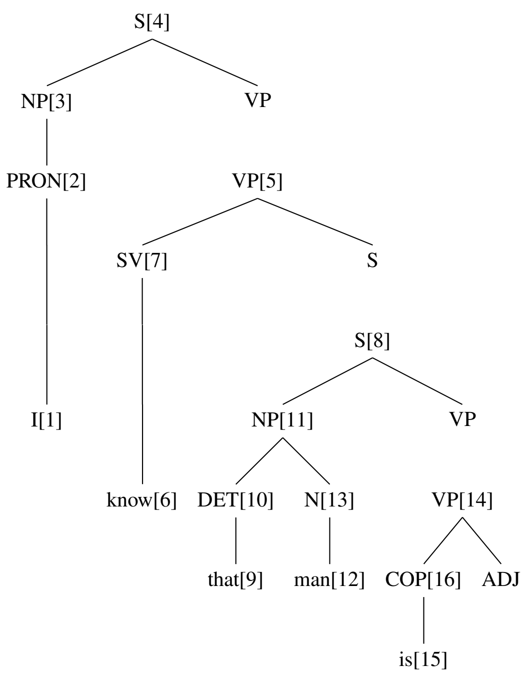
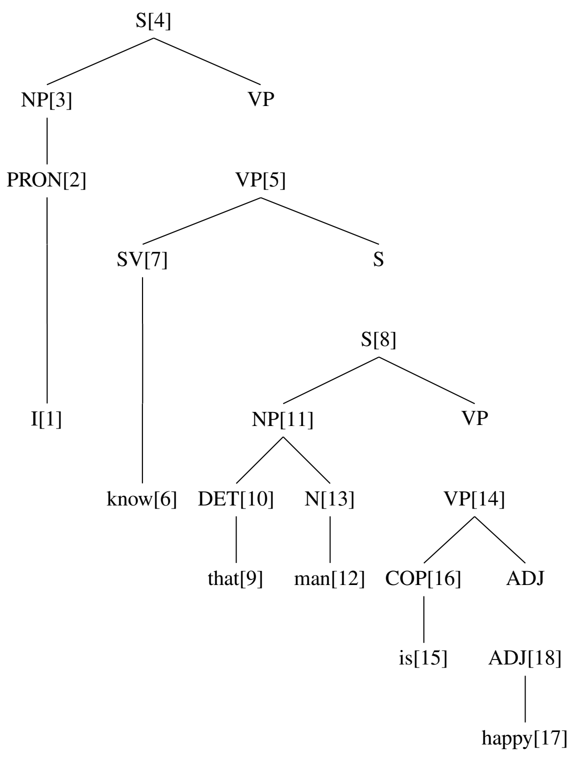


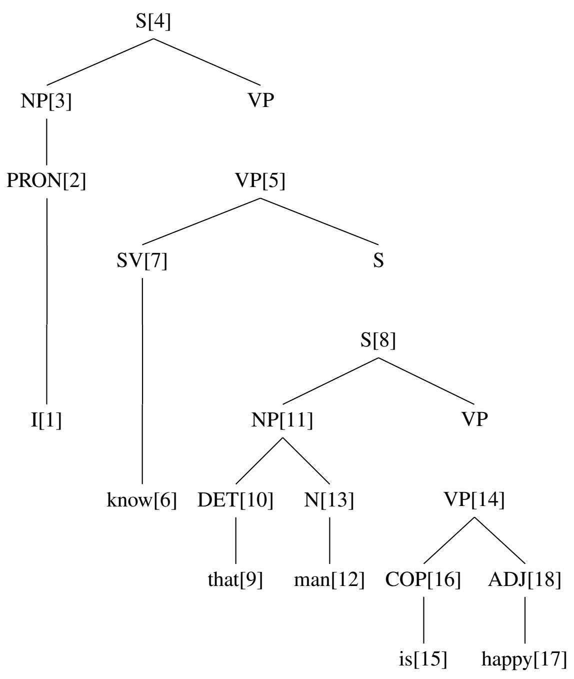
Figure A.3: Left-corder parse trees, from tree (11) to tree (14).



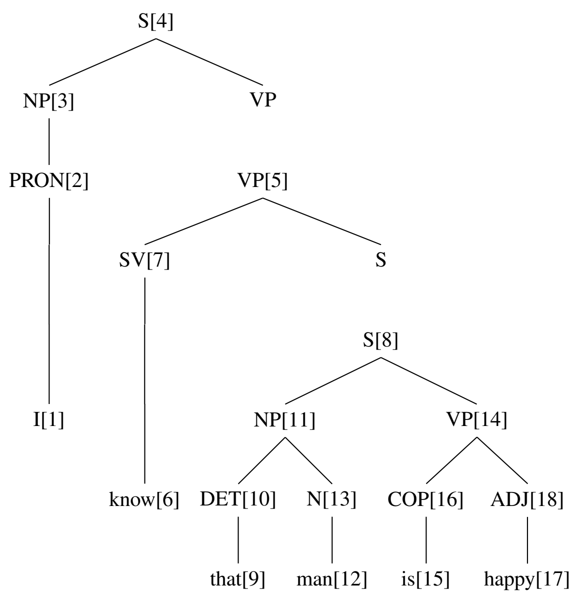
(15)



(16)

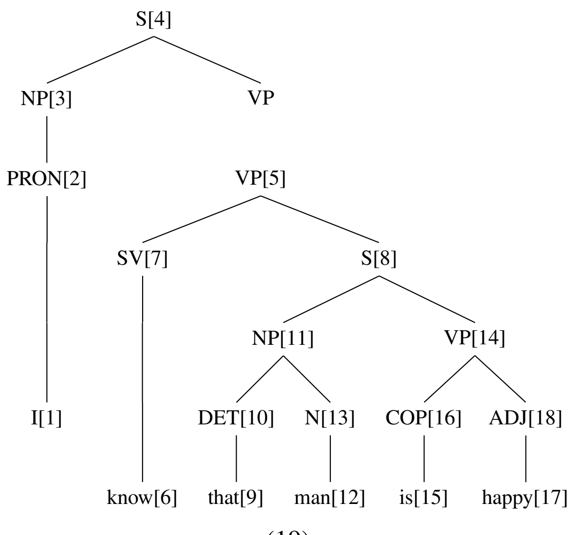


(17)

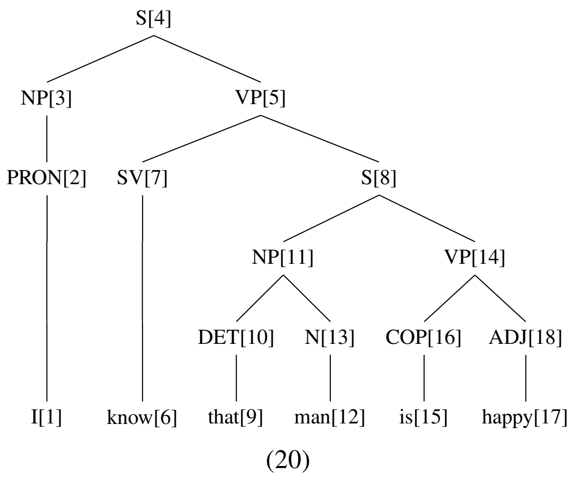


(18)

Figure A.3: Left-corder parse trees, from tree (15) to tree (18).



(19)



(20)

Figure A.3: Left-corder parse trees (19) and (20).

Appendix B

Shift-reduce Parsing Transitions

```
Step  Queue                          Stack                Action                    Alternatives
----------------------------------------------------------------------------------------------------
1     I know that man is happy       []                   Reduce: PRON → I          -
2     PRON know that man is happy    []                   Reduce: NP → PRON         -
3     NP know that man is happy      []                   Shift                     -
4     know that man is happy         NP                   Reduce: TV → know         SV → know
5     TV that man is happy           NP                   Shift                     -
6     that man is happy              TV NP                Reduce: PRON → that       DET → that
7     PRON man is happy              TV NP                Reduce: NP → PRON         -
8     NP man is happy                TV NP                Reduce: VP → TV NP        -
9     VP man is happy                NP                   Reduce: S → NP VP         -
10    S man is happy                 []                   Shift                     -
11    man is happy                   S                    Reduce: TV → man          N → man
12    TV is happy                    S                    Shift                     -
13    is happy                       TV S                 Reduce: COP → is          -
14    COP happy                      TV S                 Shift                     -
15    happy                          COP TV S             Reduce: ADJ → happy       -
16    ADJ                            COP TV S             Reduce: VP → COP ADJ      -
17    VP                             TV S                 Shift                     -
18    []                             VP TV S              Backtrack to step 11      -
19    man is happy                   S                    Reduce: N → man           -
20    N is happy                     S                    Shift                     -
21    is happy                       N S                  Reduce: COP → is          -
22    COP happy                      N S                  Shift                     -
23    happy                          COP N S              Reduce: ADJ → happy       -
24    ADJ                            COP N S              Reduce: VP                -
25    VP                             N S                  Shift                     -
26    []                             VP N S               Backtrack to step 6       -
27    that man is happy              TV NP                Reduce: DET → that        -
28    DET man is happy               TV NP                Shift                     -
29    man is happy                   DET TV NP            Reduce: TV → man          N → man
30    TV is happy                    DET TV NP            Shift                     -
31    is happy                       TV DET TV NP         Reduce: COP → is          -
32    COP happy                      TV DET TV NP         Shift                     -
33    happy                          COP TV DET TV NP     Reduce: ADJ → happy       -
34    ADJ                            COP TV DET TV NP     Reduce: VP → COP ADJ      -
35    VP                             TV DET TV NP         Shift                     -
36    []                             VP TV DET TV NP      Backtrack to step 29      -
37    man is happy                   DET TV NP            Reduce: N → man           -
38    N is happy                     DET TV NP            Reduce: NP DET N          -
39    NP is happy                    TV NP                Reduce: VP → TV NP        -
40    VP is happy                    NP                   Reduce: S → NP VP         -
```

Figure B.1: Complete parse transitions, from step 1 to step 40.

```
Step  Queue                          Stack                Action                    Alternatives
----------------------------------------------------------------------------------------------------
41    S is happy                     []                   Shift                     -
42    is happy                       S                    Reduce: COP → is          -
43    COP happy                      S                    Shift                     -
44    happy                          COP S                Reduce: ADJ → happy       -
45    ADJ                            COP S                Reduce: VP → COP ADJ      -
46    VP                             S                    Shift                     -
47    []                             VP S                 Backtrack to step 4       -
48    know that man is happy         NP                   Reduce: SV → know         -
49    SV that man is happy           NP                   Shift                     -
50    that man is happy              SV NP                Reduce: PRON → that       DET → that
51    PRON man is happy              SV NP                Reduce: NP → PRON         -
52    NP man is happy                SV NP                Shift                     -
53    man is happy                   NP SV NP             Reduce: TV → man          N - man
54    TV is happy                    NP SV NP             Shift                     -
55    is happy                       TV NP SV NP          Reduce: COP → happy       -
56    COP happy                      TV NP SV NP          Shift                     -
57    happy                          COP TV NP SV NP      Reduce: ADJ → happy       -
58    ADJ                            COP TV NP SV NP      Reduce: VP → COP ADJ      -
59    VP                             TV NP SV NP          Shift                     -
60    []                             VP TV NP SV NP       Backtrack to step 53      -
61    man is happy                   NP SV NP             Reduce N → man            -
62    N is happy                     NP SV NP             Shift                     -
63    is happy                       N NP SV NP           Reduce: COP → is          -
64    COP happy                      N NP SV NP           Shift                     -
65    happy                          COP N NP SV NP       Reduce: ADJ → happy       -
66    ADJ                            COP N NP SV NP       Reduce: VP → COP ADJ      -
67    VP                             N NP SV NP           Shift                     -
68    []                             VP N NP SV NP        Backtrack to step 50      -
69    that man is happy              SV NP                Reduce: DET → that        -
70    DET man is happy               SV NP                Shift                     -
71    man is happy                   DET SV NP            Reduce: TV - man          N → man
72    TV is happy                    DET SV NP            Shift                     -
73    is happy                       TV DET SV NP         Reduce: COP → is          -
74    COP happy                      TV DET SV NP         Shift                     -
75    happy                          COP TV DET TV NP     Reduce: ADJ → happy       -
76    ADJ                            COP TV DET TV NP     Reduce: VP → COP ADJ      -
77    VP                             TV DET TV NP         Shift                     -
78    []                             VP TV DET TV NP      Backtrack to step 71      -
79    man is happy                   DET SV NP            Reduce: N → man           -
80    N is happy                     DET SV NP            Reduce: NP → DET N        -
81    NP is happy                    SV NP                Shift                     -
82    is happy                       NP SV NP             Reduce: COP → is          -
83    COP happy                      NP SV NP             Shift                     -
84    happy                          COP NP SV NP         Reduce: ADJ → happy       -
85    ADJ                            COP NP SV NP         Reduce: VP → COP ADJ      -
86    VP                             NP SV NP             Reduce: S → NP VP         -
87    S                              SV NP                Reduce: VP → SV S         -
88    VP                             NP                   Reduce: S → NP VP         -
89    S                              []                   Shift                     -
90    []                             S                    -                         -
```

Figure B.2: Remaining parse transitions.

Appendix C

CoNLL-X data file format

The data in the CoNLL file follows the following rules:

- A blank line marks the start of a new sentence.
- Each sentence consists of one or more tokens and each token starts on a new line.
- For each token, ten fields are used for describing it [1], as shown in Table C.1. A single tab character is used for separating each field while Space/blank characters are not allowed between fields.
- For each non-dummy values for the fields (ID, FORM, CPOSTAG, POSTAG, HEAD and DEPREL) a non-underscore value is used in order to guarantee that that the file contains non-dummy values for all languages.
- UTF-8 unicode is used for encoding the data files.

An example of the dependency tree for the sentence *John loves Mary* is shown in Figure C.1 and the CoNLL representation for it is shown in Figure C.2 .

[1] See http://ilk.uvt.nl/conll/#dataformat.

#	Field name	Description
1	ID	Token counter, starting at 1 for each new sentence.
2	FORM	Word form or punctuation symbol.
3	LEMMA	Lemma or stem (depending on particular data set) of word form, or an underscore if not available.
4	CPOSTAG	Coarse-grained POS tag, where tagset depends on the language.
5	POSTAG	Fine-grained POS tag, where the tagset depends on the language, or identical to the coarse-grained part-of-speech tag if not available.
6	FEATS	Unordered set of syntactic and/or morphological features (depending on the particular language), separated by a vertical bar (|), or an underscore if not available.
7	HEAD	Head of the current token, which is either a value of ID or zero ('0'). Note that depending on the original treebank annotation, there may be multiple tokens with an ID of zero.
8	DEPREL	Dependency relation to the HEAD. The set of dependency relations depends on the particular language. Note that depending on the original treebank annotation, the dependency relation may be meaningful or simply 'ROOT'.
9	PHEAD	Projective head of current token, which is either a value of ID or zero ('0'), or an underscore if not available. Note that depending on the original treebank annotation, there may be multiple tokens an with ID of zero. The dependency structure resulting from the PHEAD column is guaranteed to be projective (but is not available for all languages), whereas the structures resulting from the HEAD column will be non-projective for some sentences of some languages (but is always available).
10	PDEPREL	Dependency relation to the PHEAD, or an underscore if not available. The set of dependency relations depends on the particular language. Note that depending on the original treebank annotation, the dependency relation may be meaningful or simply 'ROOT'.

Table C.1: CoNLL-X data file format.

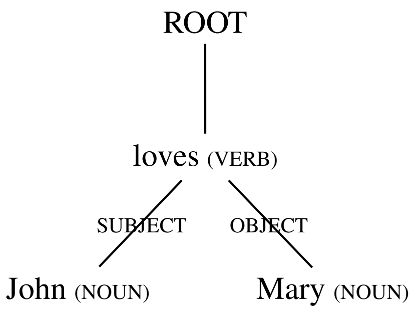


Figure C.1: Dependency tree for the sentence *John loves Mary*.

```
0    root    _    ROOT    ROOT    _    _    ROOT       _    _
1    John    _    NOUN    NOUN    _    2    SUBJECT    _    _
2    loves   _    VERB    VERB    _    0    ROOT       _    _
3    Mary    _    NOUN    NOUN    _    2    OBJECT     _    _
```

Figure C.2: CoNLL format for the sentence *John loves Mary*.

Appendix D

The Variation of HPTs

This appendix contains the different versions of the Head Percolation Tables (HPT) that we have mentioned in Chapter 5 Section 5.1.3.2.

The organisation of the head child in the HPT V.1, as in Figure D.1 is performed randomly during the conversion of the Penn Arabic Treebank from phrase structure format to dependency format.

The HPT V.2, as in Figure D.2, is similar to HPT V.1 but In this version coordinate conjunctions (CONJ) into CONJ and ICONJ, where ICONJ is used for sentence-initial coordinate and CONJ is used for non-sentence-initial coordinate.

The HPT V.3, as in Figure D.3, is similar to HPT V.2 but items in the entries are ordered according to their phrase category. For example, adjectives in adjectival phrases are put before nouns, prepositions etc. Additionally, coordinating conjunctions (CONJ and ICONJ) have the highest priority inside all the entries and determiners are given higher priority than nouns.

HPT V4 is the same as HPT V.3 but coordinating conjunctions (CONJ and ICONJ) have the lowest possible priority inside all the entries.

HPT V.5 nouns are given a higher priority than determiners and everything else is the same as HPT V.3.

```
POS tag     Possible head-child(s)
----------------------------------------------------------------------------------------------------
ADJP:       ADJP, ADV, ADJ, DET+ADJ, NOUN_NUM, PP, NOUN, DET+NOUN, PRON, PRN, DET, PRT, PART, NP,
            NUM, CONJP, CONJ, PREP
ADJP-OBJ:   ADJ, NOUN
ADJP-PRD:   ADV, ADJP, ADJ, PP, NOUN, SBAR, PRON, PRT, PART, NP, CONJ, PREP
ADV:        ADV, ADJP, ABBREV, ADJP-PRD, ADJ, DET+ADJ, SBAR, DET, PART, CONJ, PP, NOUN, PRN, PRT,
            NUM, DET+NOUN, NP, QP, NOUN_PROP, VP, PV, S, VERB, PUNC, NOUN_NUM, NP-TPC, PP-PRD,
            NP-SBJ, WHNP, PRON, SUB_CONJ, PREP
CONJP:      CONJ, ADV, NOUN, PART, ADJ, PREP
FRAG:       FRAG, ADV, PP, NOUN, SBAR, PRN, PRT, PART, SUB_CONJ, NP, CONJ, ADJ
NAC:        NOUN, NP, ADV, PP, SBAR, UCP, ADJP, CONJP, PRT, S, SUB_CONJ, CONJ, PREP
NP:         NP, NAC, NOUN, DET+NOUN, DET+NUM, NP-OBJ, NOUN_NUM, NOUN_PROP, NP-SBJ, SBAR, ADJP,
            DET+ADJ, DET, CONJP, PART, INTERJ, CONJ, WHNP, PP, LATIN, UCP, FRAG, PRN, PRT, ABBREV,
            PV, ADV, IV, VP, S, X, ADJ, QP, PUNC, PRON, SUB_CONJ, PREP
NP-OBJ:     NP-OBJ, NAC, NOUN, DET+NOUN, NUM, NP, NOUN_NUM, NOUN_PROP, NP-TPC, SBAR, ADJP, CONJP,
            PART, INTERJ, X, CONJ, PP, PRN, PRT, ABBREV, ADV, IV, VP, S, PV, ADJ, QP, PRON, PREP
NP-PRD:     NOUN, NOUN_PROP, DET+NOUN, NP, NUM, NOUN_NUM, ADV, QP, PP, PV, SBAR, PRT, ADJP, PRN,
            CONJP, DET, PRON, ABBREV, PART, UCP, CONJ, ADJ, PREP
NP-SBJ:     NP-SBJ, NAC, NP-PRD, NOUN, DET+NOUN, NUM, NP, NP-OBJ, NOUN_PROP, NOUN_NUM, FRAG, SBAR,
            CONJP, PART, X, CONJ, PP, UCP, PRN, PRT, ADJP, ADV, QP, ADJP-PRD, S, PV, ADJ, DET+ADJ,
            PP-PRD, PRON, PREP
NP-TPC:     NAC, NUM, NP, NOUN_NUM, NOUN_PROP, NOUN, SBAR, ADJP, X, CONJ, PP, PV, UCP, PRN, PRT,
            ABBREV, ADV, ADJ, QP, PUNC, PRON, PREP
PP:         PP, PART, PRN, PRT, PV, PRON, PREP, NAC, SBAR, ADJP, CONJP, X, CONJ, NOUN, DET+NOUN,
            UCP, NEG_PART, FRAG, ABBREV, NUM, NP, NP-OBJ, ADV, QP, NOUN_PROP, VP, S, ADJ, NP-TPC,
            NP-SBJ, WHNP, SUB_CONJ
PP-OBJ:     PREP, PP, NP, NOUN
PP-PRD:     PP, PRT, PRN, PRON, PART, PREP, ADV, QP, NOUN, SBAR, NP-SBJ, ADJP, CONJP, S, SUB_CONJ,
            NP, X, CONJ, ADJ
PP-SBJ:     PREP, NP
PRN:        PP, PART, PUNC, ADV, NOUN, DET+NOUN, NO_FUNC, SBAR, ADJP, ABBREV, S, NUM, NOUN_PROP, NP,
            CONJ, ADJ
PRT:        PRT, VERB, CONJ
QP:         NOUN_NUM, NOUN, NUMERIC_COMMA, DET, PART, ABBREV, NUM, CONJ, ADJ, PREP
S:          S, S-PRD, SBAR, S-SBJ, SQ, SBARQ-PRD, SUB_CONJ, FRAG, FRAG-PRD, NAC, NP-PRD, CONJP, PART,
            PP-SBJ, X, CONJ, PP, PV, ADJP-PRD, INTJ, UCP, PRN, PRT, NUM, ADJP, NP, ADV, UCP-PRD, IV,
            VERB, VP, LATIN, PUNC, UCP-SBJ, ADJ, NP-TPC, NO_FUNC, INTJ-PRD, PP-PRD, NP-SBJ
S-PRD:      ADV, NP-TPC, PP, VP
S-SBJ:      SUB_CONJ, VP
SBAR:       SBAR, SUB, S, SUB_CONJ, ADV, PP, PV, VERB, PRT, PUNC, INTERJ, UCP, WHNP, PRON, PART, X,
            NOUN, CONJ, CONJP, PREP, VERB
SBAR-SBJ:   SBAR,  SUB_CONJ, S
SBARQ:      SQ, S, ADV
SBARQ-PRD:  S, ADV
SQ:         NP-SBJ, PP-PRD
UCP:        ADV, FRAG, PP, SBAR, ADJP, VP, S, NP, CONJ
UCP-OBJ:    NP, CONJ, SBAR
UCP-PRD:    NP, ADJP, CONJ, S, PP
UCP-SBJ:    ADV, PP, SBAR, VP, NP, CONJ
VP:         VP, VERB, NAC, SBAR, NP-PRD, CONJP, ADJP-OBJ, PART, PP-SBJ, X, CONJ, WHNP,  PP, PV, UCP,
            FRAG, PRN, PRT, ADJP-PRD, NP, NP-OBJ, UCP-OBJ, ADV, UCP-PRD, NOUN_PROP, PP-OBJ, IV, S, NOUN,
            UCP-SBJ, ADJ, CV, NP-TPC, PP-PRD, NP-SBJ, SBARQ, PUNC, ADJP, PRON, SUB_CONJ, PREP
WHNP:       WHNP, ADV, NOUN, PRON, NP, CONJ, PP, PREP
X:          LATIN, NOUN, NO_FUNC, PUNC, NUM, IV, PRON, ABBREV, PART, NOUN_PROP, SUB_CONJ, NEG_PART, NP,
            PV, CONJ, PREP
```

Figure D.1: HPT V.1: randomly organised head child.

POS tag	Possible head-child(s)
ADJP:	ADJP, ADV, ADJ, DET+ADJ, NOUN_NUM, PP, NOUN, DET+NOUN, PRON, PRN, DET, PRT, PART, NP, NUM, CONJP, CONJ, PREP
ADJP-OBJ:	ADJ, NOUN
ADJP-PRD:	ADV, ADJP, ADJ, PP, NOUN, SBAR, PRON, PRT, PART, NP, CONJ, PREP
ADV:	ADV, ADJP, ABBREV, ADJP-PRD, ADJ, DET+ADJ, SBAR, DET, PART, CONJ, PP, NOUN, PRN, PRT, NUM, DET+NOUN, NP, QP, NOUN_PROP, VP, PV, S, VERB, PUNC, NOUN_NUM, NP-TPC, PP-PRD NP-SBJ, WHNP, PRON, SUB_CONJ, PREP
CONJP:	CONJ, ADV, NOUN, PART, ADJ, PREP
FRAG:	FRAG, ADV, PP, NOUN, SBAR, PRN, PRT, PART, SUB_CONJ, NP, CONJ, ADJ
NAC:	NOUN, NP, ADV, PP, SBAR, UCP, ADJP, CONJP, PRT, S, SUB_CONJ, CONJ, PREP
NP:	NP, NAC, NOUN, DET+NOUN, NUM, DET+NUM, NP-OBJ, NOUN_NUM, NOUN_PROP, NP-SBJ, SBAR, ADJP, DET+ADJ, DET, CONJP, PART, INTERJ, CONJ, WHNP, PP, LATIN, UCP, FRAG, PRN, PRT, ABBREV, PV, ADV, IV, VP, S, X, ADJ, QP, PUNC, PRON, SUB_CONJ, PREP
NP-OBJ:	NP-OBJ, NAC, NOUN, DET+NOUN, NUM, NP, NOUN_NUM, NOUN_PROP, NP-TPC, SBAR, ADJP, CONJP, PART, INTERJ, X, CONJ, PP, PRN, PRT, ABBREV, ADV, IV, VP, S, PV, ADJ, QP, PRON, PREP
NP-PRD:	NOUN, NOUN_PROP, DET+NOUN, NP, NUM, NOUN_NUM, ADV, QP, PP, PV, SBAR, PRT, ADJP, PRN, CONJP, DET, PRON, ABBREV, PART, UCP, CONJ, ADJ, PREP
NP-SBJ:	NP-SBJ, NAC, NP-PRD, NOUN, DET+NOUN, NUM, NP, NP-OBJ, NOUN_PROP, NOUN_NUM, FRAG, SBAR, CONJP, PART, X, CONJ, PP, UCP, PRN, PRT, ADJP, ADV, QP, ADJP-PRD, S, PV, ADJ, DET+ADJ, PP-PRD, PRON, PREP
NP-TPC:	NAC, NUM, NP, NOUN_NUM, NOUN_PROP, NOUN, SBAR, ADJP, X, CONJ, PP, PV, UCP, PRN, PRT, ABBREV, ADV, ADJ, QP, PUNC, PRON, PREP
PP:	PP, PART, PRN, PRT, PV, PRON, PREP, NAC, SBAR, ADJP, CONJP, X, CONJ, NOUN, DET+NOUN, UCP, NEG_PART, FRAG, ABBREV, NUM, NP, NP-OBJ, ADV, QP, NOUN_PROP, VP, S, ADJ, NP-TPC, NP-SBJ, WHNP, SUB_CONJ
PP-OBJ:	PREP, PP, NP, NOUN
PP-PRD:	PP, PRT, PRN, PRON, PART, PREP, ADV, QP, NOUN, SBAR, NP-SBJ, ADJP, CONJP, S, SUB_CONJ, NP, X, CONJ, ADJ
PP-SBJ:	PREP, NP
PRN:	PP, PART, PUNC, ADV, NOUN, DET+NOUN, NO_FUNC, SBAR, ADJP, ABBREV, S, NUM, NOUN_PROP, NP, CONJ, ADJ
PRT:	PRT, VERB, ICONJ
QP:	NOUN_NUM, NOUN, NUMERIC_COMMA, DET, PART, ABBREV, NUM, CONJ, ADJ, PREP
S:	S, S-PRD, SBAR, S-SBJ, SQ, SBARQ-PRD, SUB_CONJ, FRAG, FRAG-PRD, NAC, NP-PRD, CONJP, PART, PP-SBJ, X, CONJ, ICONJ, PP, PV, ADJP-PRD, INTJ, UCP, PRN, PRT, NUM, ADJP, NP, ADV, UCP-PRD, IV, VERB, VP, LATIN, PUNC, UCP-SBJ, ADJ, NP-TPC, NO_FUNC, INTJ-PRD, PP-PRD, NP-SBJ
S-PRD:	ADV, NP-TPC, PP, VP
S-SBJ:	SUB_CONJ, VP
SBAR:	SBAR, SUB, S, SUB_CONJ, ADV, PP, PV, VERB, PRT, PUNC, INTERJ, UCP, WHNP, PRON, PART, X, NOUN, ICONJ, CONJP, PREP, VERB
SBAR-SBJ:	SBAR, SUB_CONJ, S
SBARQ:	SQ, S, ADV
SBARQ-PRD:	S, ADV
SQ:	NP-SBJ, PP-PRD
UCP:	ADV, FRAG, PP, SBAR, ADJP, VP, S, NP, CONJ
UCP-OBJ:	NP, CONJ, SBAR
UCP-PRD:	NP, ADJP, CONJ, S, PP
UCP-SBJ:	ADV, PP, SBAR, VP, NP, CONJ
VP:	VP, VERB, NAC, SBAR, NP-PRD, CONJP, ADJP-OBJ, PART, PP-SBJ, X, CONJ, WHNP, ICONJ, PP, PV, UCP, FRAG, PRN, PRT, ADJP-PRD, NP, NP-OBJ, UCP-OBJ, ADV, UCP-PRD, NOUN_PROP, PP-OBJ, IV, S, NOUN, UCP-SBJ, ADJ, CV, NP-TPC, PP-PRD, NP-SBJ, SBARQ, PUNC, ADJP, PRON, SUB_CONJ, PREP
WHNP:	WHNP, ADV, NOUN, PRON, NP, CONJ, PP, PREP
X:	LATIN, NOUN, NO_FUNC, PUNC, NUM, IV, PRON, ABBREV, PART, NOUN_PROP, SUB_CONJ, NEG_PART, NP, PV, CONJ, PREP, ICONJ

Figure D.2: HPT V.2.

POS tag	Possible head-child(s)
ADJP	CONJ, CONJP, ADJ, DET+ADJ, ADJP, DET, DET+NOUN, NOUN, NP
ADJP-OBJ	ADJ
ADJP-PRD	CONJ, ADJ, ADJP, NOUN, NP
ADV	CONJ, ADV, VP, PV, VERB, S, DET+NOUN, NOUN, NP, NP-SBJ, REL_PRON, PRON, NOUN_NUM, PART, ADJ, DET+ADJ, PP, NUM, SUB_CONJ, SBAR
CONJP	ICONJ, CONJ, NOUN, ADJ, PART, PREP
FRAG	CONJ, SBAR, NOUN, NP, PRT, ADV, PP, FRAG
NAC	CONJ, NOUN, NP, ADV, SUB_CONJ
NP	ICONJ, CONJ, NOUN, DET, DET+NOUN, NOUN_PROP, NP, POSS_PRON, DEM_PRON, PRON, PV, SBAR, PART, PRN, ADV, ADJ, DET+ADJ, PREP, PP, QP, NOUN_NUM, NUM, ABBREV, VP, S, FOREIGN
NP-OBJ	CONJ, DET, DET+NOUN, NOUN, NOUN_PROP, NP, PRON, VP, S, ADJ, DET+ADJ, QP, NOUN_NUM, NUM, ABBREV
NP-PRD	CONJ, NOUN, NOUN_PROP, NP, PRON, PART, ADJ, QP, NOUN_NUM, NUM, ABBREV
NP-SBJ	CONJ, DET, DET+NOUN, NOUN, NOUN_PROP, NP, PRON, SBAR, PRT, PV, ADJ, DET+ADJ, QP, ABBREV
NP-TPC	CONJ, NOUN, NOUN_PROP, NP, PRON, ADV, ADJ, NUM, PUNC
PP	ICONJ, CONJ, PREP, PP, DET+NOUN, NOUN, NOUN_PROP, SBAR, PV, VP, X, NP, S, NEG_PART
PP-OBJ	PREP, PP, NOUN
PP-PRD	CONJ, PREP, PP, NOUN, NP, S, PART, SBAR
PP-SBJ	PREP
PRN	CONJ, S, NP, ADV, PP, DET+NOUN, NOUN, NOUN_PROP, NUM, SBAR, ADJ, ADJP
PRT	ICONJ, VERB, PART, PRT
QP	CONJ, NOUN, NOUN_NUM, NUM, ADJ
S	ICONJ, S, VERB, PV, IV, VP, CONJP, CONJ, NP, NP-TPC, NP-SBJ, NP-PRD, PRT, SBAR, ADV, ADJP, ADJP-PRD, PP, PP-PRD, PART, FRAG, FRAG-PRD, X
S-PRD	VP
S-SBJ	VP
SBAR	ICONJ, CONJ, VERB, PV, INTERJ, NOUN, S, PART, PRON, SBAR, UCP, SUB_CONJ, WHNP
SBAR-SBJ	SBAR, S, SUB_CONJ
SBARQ	SQ, S
SBARQ-PRD	S
SQ	NP-SBJ
UCP	CONJ, S, NP, SBAR, VP, ADV, ADJP
UCP-OBJ	CONJ, NP
UCP-PRD	CONJ, NP
UCP-SBJ	CONJ, NP
VP	ICONJ, CONJ, PRT, VERB, PV, IV, CV, VP, PART, X, S, SBAR, NOUN, NOUN_PROP, NP, NP-SBJ, NP-PRD, NP-OBJ, NP-TPC, ADJ , ADV, PREP, PP, NEG_PART, NAC
WHNP	CONJ, NOUN, NP, PRON, PREP, PP, WHNP
X	ICONJ, CONJ, PV, IV, PART, NOUN, NP, NOUN_PROP, PRON, PREP, NEG_PART, NUM, ABBREV, NO_FUNC, PUNC

Figure D.3: HPT V.3.

```
POS tag     Possible head-child(s)
----------------------------------------------------------------------------------------------------
ADJP        ADJ, DET+ADJ, ADJP, DET, NOUN, DET+NOUN, NP, CONJ, CONJP
ADJP-OBJ    ADJ
ADJP-PRD    ADJ, ADJP, NOUN, NP, CONJ
ADV         ADV, VP, PV, VERB, S,  DET+NOUN, NOUN, NP, NP-SBJ, REL_PRON, PRON, NOUN_NUM, PART,
            ADJ, DET+ADJ, PP,  NUM, SUB_CONJ, SBAR, CONJ
CONJP       NOUN, ADJ, PART, PREP, ICONJ, CONJ
FRAG        SBAR, NOUN, NP, PRT, ADV, PP, FRAG, CONJ
NAC         NOUN, NP, ADV, SUB_CONJ, CONJ
NP          DET, DET+NOUN, NOUN, NOUN_PROP, NP, POSS_PRON, DEM_PRON, PRON, PV, SBAR, PART, PRN,
            ADV, ADJ, DET+ADJ, PREP, PP, QP, NOUN_NUM, NUM, ABBREV, VP, S, FOREIGN, ICONJ, CONJ
NP-OBJ      DET, DET+NOUN, NOUN, NOUN_PROP, NP, PRON, VP, S, ADJ, DET+ADJ, QP, NOUN_NUM, NUM, ABBREV,
            CONJ
NP-PRD      NOUN, NOUN_PROP, NP, PRON, PART, ADJ, QP, NOUN_NUM, NUM, ABBREV, CONJ
NP-SBJ      DET, DET+NOUN, NOUN, NOUN_PROP, NP, PRON, SBAR, PRT, PV, ADJ, DET+ADJ, QP, ABBREV,
            CONJ
NP-TPC      NOUN, NOUN_PROP, NP, PRON, ADV, ADJ, NUM, PUNC, CONJ
PP          PREP, PP, DET+NOUN, NOUN, NOUN_PROP, SBAR, PV, VP, X, NP, S, NEG_PART, ICONJ, CONJ
PP-OBJ      PREP, PP, NOUN
PP-PRD      PREP, PP, NOUN, NP, S, PART, SBAR, CONJ
PP-SBJ      PREP
PRN         S, NP, ADV, PP, DET+NOUN, NOUN, NOUN_PROP, NUM, SBAR, ADJ, ADJP, CONJ
PRT         VERB, PART, PRT, ICONJ
QP          NOUN, NOUN_NUM, NUM, ADJ, CONJ
S           S, VERB, PV, IV, VP, CONJP, CONJ, NP, NP-TPC, NP-SBJ, NP-PRD, PRT, SBAR, ADV, ADJP,
            ADJP-PRD
            PP, PP-PRD, PART, FRAG, FRAG-PRD, X, ICONJ
S-PRD       VP
S-SBJ       VP
SBAR        VERB, PV, INTERJ, NOUN, S, PART, PRON, SBAR, UCP, SUB_CONJ, WHNP, ICONJ, CONJ
SBAR-SBJ    SBAR, S, SUB_CONJ
SBARQ       SQ, S
SBARQ-PRD   S
SQ          NP-SBJ
UCP         S, NP, SBAR, VP, ADV, ADJP, CONJ
UCP-OBJ     NP ,CONJ
UCP-PRD     NP, CONJ
UCP-SBJ     NP, CONJ
VP          ICONJ, CONJ, PRT, VERB, PV, IV, CV, VP, PART, X, S, SBAR, NOUN, NOUN_PROP, NP, NP-SBJ,
            NP-PRD, NP-OBJ, NP-TPC, ADJ , ADV, PREP, PP, NEG_PART, NAC, ICONJ, CONJ
WHNP        NOUN, NP, PRON, PREP, PP, WHNP, CONJ
X           PV, IV, PART, NOUN, NP, NOUN_PROP, PRON, PREP, NEG_PART, NUM, ABBREV, NO_FUNC, PUNC,
            ICONJ, CONJ
```

Figure D.4: HPT V.4.

POS tag	Possible head-child(s)
ADJP	CONJ, CONJP, ADJ, DET+ADJ, ADJP, NOUN, DET+NOUN, DET, NP
ADJP-OBJ	ADJ
ADJP-PRD	CONJ, ADJ, ADJP, NOUN, NP
ADV	CONJ, ADV, VP, PV, VERB, S, NOUN, DET+NOUN, NP, NP-SBJ, REL_PRON, PRON, NOUN_NUM, PART, ADJ, DET+ADJ, PP, NUM, PART, SUB_CONJ, SBAR
CONJP	ICONJ, CONJ, NOUN, ADJ, PART, PREP
FRAG	CONJ, SBAR, NOUN, NP, PRT, ADV, PP, FRAG
NAC	CONJ, NOUN, NP, ADV, SUB_CONJ
NP	ICONJ, CONJ, NOUN, NOUN_PROP, DET+NOUN, DET, NP, POSS_PRON, DEM_PRON, PRON, PV, SBAR, PART, PRN, ADV, ADJ, DET+ADJ, PREP, PP, QP, NOUN_NUM, NUM, ABBREV, VP, S, FOREIGN
NP-OBJ	CONJ, NOUN, NOUN_PROP, DET+NOUN, DET, NP, PRON, VP, S, ADJ, DET+ADJ, QP, NOUN_NUM, NUM, ABBREV
NP-PRD	CONJ, NOUN, NOUN_PROP, NP, PRON, PART, ADJ, QP, NOUN_NUM, NUM, ABBREV
NP-SBJ	CONJ, NOUN, NOUN_PROP, DET+NOUN, DET, NP, PRON, SBAR, PRT, PV, ADJ, DET+ADJ, QP, ABBREV
NP-TPC	CONJ, NOUN, NOUN_PROP, NP, PRON, ADV, ADJ, NUM, PUNC
PP	ICONJ, CONJ, PREP, PP, NOUN, DET+NOUN, NOUN_PROP, SBAR, PV, VP, X, NP, S, NEG_PART
PP-OBJ	PREP, PP, NOUN
PP-PRD	CONJ, PREP, PP, NOUN, NP, S, PART, SBAR
PP-SBJ	PREP
PRN	CONJ, S, NP, ADV, PP, NOUN, NOUN_PROP, DET+NOUN, NUM, SBAR, ADJ, ADJP
PRT	ICONJ, VERB, PART, PRT
QP	CONJ, NOUN, NOUN_NUM, NUM, ADJ
S	ICONJ, S, VERB, PV, IV, VP, CONJP, CONJ, NP, NP-TPC, NP-SBJ, NP-PRD, PRT, SBAR, ADV, ADJP, ADJP-PRD, PP, PP-PRD, PART, FRAG, FRAG-PRD, X
S-PRD	VP
S-SBJ	VP
SBAR	ICONJ, CONJ, VERB, PV, INTERJ, NOUN, S, PART, PRON, SBAR, UCP, SUB_CONJ, WHNP
SBAR-SBJ	SBAR, S, SUB_CONJ
SBARQ	SQ, S
SBARQ-PRD	S
SQ	NP-SBJ
UCP	CONJ, S, NP, SBAR, VP, ADV, ADJP
UCP-OBJ	CONJ, NP
UCP-PRD	CONJ, NP
UCP-SBJ	CONJ, NP
VP	ICONJ, CONJ, PRT, VERB, PV, IV, CV, VP, PART, X, S, SBAR, NOUN, NOUN_PROP, NP, NP-SBJ, NP-PRD, NP-OBJ, NP-TPC, ADJ , ADV, PREP, PP, NEG_PART, NAC
WHNP	CONJ, NOUN, NP, PRON, PREP, PP, WHNP
X	ICONJ, CONJ, PV, IV, PART, NOUN, NP, NOUN_PROP, PRON, PREP, NEG_PART, NUM, ABBREV, NO_FUNC, PUNC

Figure D.5: HPT V.5.