Technical Report

UCAM-CL-TR-743
ISSN 1476-2986

Number 743

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Optimising the speed and accuracy of a Statistical GLR Parser

Rebecca F. Watson

March 2009

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 2009 Rebecca F. Watson

This technical report is based on a dissertation submitted September 2007 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Darwin College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

The focus of this thesis is to develop techniques that optimise *both* the speed and accuracy of a unification-based statistical GLR parser. However, we can apply these methods within a broad range of parsing frameworks. We first aim to optimise the level of tag ambiguity resolved during parsing, given that we employ a front-end PoS tagger. This work provides the first broad comparison of tag models as we consider *both* tagging and parsing performance. A *dynamic* model achieves the best accuracy and provides a means to overcome the trade-off between tag error rates in single tag per word input and the increase in parse ambiguity over multiple-tag per word input. The second line of research describes a novel modification to the inside-outside algorithm, whereby *multiple* inside and outside probabilities are assigned for elements within the packed parse forest data structure. This algorithm enables us to compute a set of 'weighted GRs' directly from this structure. Our experiments demonstrate substantial increases in parser accuracy and throughput for weighted GR output.

Finally, we describe a novel *confidence-based* training framework, that can, in principle, be applied to any statistical parser whose output is defined in terms of its consistency with a given level and type of annotation. We demonstrate that a semisupervised variant of this framework outperforms both Expectation-Maximisation (when both are constrained by unlabelled partial-bracketing) and the extant (fully supervised) method. These novel training methods utilise data *automatically* extracted from existing corpora. Consequently, they require no manual effort on behalf of the grammar writer, facilitating grammar development.

Acknowledgements

I would first like to thank Ted Briscoe, who was an excellent supervisor. He has helped to guide this thesis with his invaluable insight and I have appreciated his patience and enthusiasm. Without his easy-going nature and constant support and direction this thesis would not have been completed as and when it was. Most importantly, he always reminded me to enjoy my time at Cambridge and have a nice glass of wine whenever possible! I would also like to thank John Carroll who even at a distance has managed to provide a great deal of support and was always available when I needed help or advice.

People from the NLIP group and administrative staff at the Computer Laboratory were also very helpful. I enjoyed my many talks with Anna Ritchie, Ben Medlock and Bill Hollingsworth. I will miss their moral support and I'm grateful that fate locked us in a room together for so many years! Thanks also to Gordon Royle and other staff at the University of Western Australia who supported me while I completed my research while visiting this University at home in Perth.

I also greatly appreciated the feedback I received during my PhD Viva. Both of my examiners, Stephen Clark and Anna Korhonen, provided helpful and thoughtful suggestions which improved the overall quality of this work's presentation.

This research would not have been possible without the financial support of both the Overseas Research Students Awards Scheme and the Poynton Scholarship awarded by the Cambridge Australia Trust in collaboration with the Cambridge Commonwealth Trust.

On a personal note I would like to thank my family; my parents and my sister Kathryn, who were always available to talk and provided a great deal of support. Finally, special thanks goes to my partner James who moved across the world to support me during my PhD. He made our home somewhere I didn't mind working on weekends.

Contents

1 Introduction **13**
1.1 Natural Language Parsing . . . 13
1.1.1 Problem Definition . . . 13
1.1.2 Corpus-based Estimation . . . 14
1.1.3 Statistical Approaches . . . 15
1.2 Research Background . . . 18
1.3 Available Resources . . . 18
1.3.1 Corpora . . . 18
1.3.2 Evaluation . . . 23
1.4 Research Goals . . . 26
1.5 Thesis Summary . . . 26
1.5.1 Contributions of this Thesis . . . 26
1.5.2 Outline of Subsequent Chapters . . . 27

2 LR Parsers **28**
2.1 Introduction . . . 28
2.2 Finite Automata . . . 29
2.2.1 NFA . . . 29
2.2.2 DFA . . . 31
2.3 LR Parsers . . . 32
2.3.1 LR Parsing Model . . . 33
2.3.2 Types of LR Parsers . . . 34
2.3.3 Parser Actions . . . 34
2.3.4 LR Table . . . 35
2.3.5 Parsing Program . . . 38
2.3.6 Table Construction . . . 38
2.4 GLR Parsing . . . 43
2.4.1 Relationship to the LR Parsing Framework . . . 43
2.4.2 Table Construction . . . 43
2.4.3 Graph-structured Stack . . . 44
2.4.4 Parse Forest . . . 47
2.4.5 LR Parsing Program . . . 47
2.4.6 Output . . . 50
2.4.7 Modifications to the Algorithm . . . 50
2.5 Statistical GLR (SGLR) Parsing . . . 50
2.5.1 Probabilistic Approaches . . . 51

2.5.2 Estimating Action Probabilities 51
2.6 RASP 53
2.6.1 Grammar 53
2.6.2 Training 58
2.6.3 Parser Application 58
2.6.4 Output Formats 61

3 Part-of-speech Tag Models **65**
3.1 Previous Work 65
3.1.1 PoS Taggers and Parsers 65
3.1.2 Tag Models 67
3.1.3 HMM PoS Taggers 68
3.2 RASP's Architecture 70
3.2.1 Processing Stages 70
3.2.2 PoS Tagger 70
3.3 Part-of-speech Tag Models 73
3.3.1 Part-of-speech Tag Files 73
3.3.2 Thresholding over Tag Probabilities 74
3.3.3 Top-ranked Parse Tags 75
3.3.4 Highest Count Tags 76
3.3.5 Weighted Count Tags 77
3.3.6 Gold Standard Tags 77
3.3.7 Summary 77
3.4 Part-of-speech Tagging Performance 77
3.4.1 Evaluation 77
3.4.2 Results 80
3.5 Parser Performance 81
3.5.1 Evaluation 81
3.5.2 Results 82
3.6 Discussion 84

4 Efficient Extraction of Weighted GRs **86**
4.1 Inside-Outside Algorithm (IOA) 87
4.1.1 Background 87
4.1.2 The Standard Algorithm 88
4.1.3 Extension to LR Parsers 92
4.2 Extracting Grammatical Relations 94
4.2.1 Modification to Local Ambiguity Packing 94
4.2.2 Extracting Grammatical Relations 95
4.2.3 Problem: Multiple Lexical Heads 97
4.2.4 Problem: Multiple Parse Forests 100
4.3 The EWG Algorithm 101
4.3.1 Inside Probability Calculation and GR Instantiation 102
4.3.2 Outside Probability Calculation 105
4.3.3 Related Work 107
4.4 EWG Performance 107
4.4.1 Comparing Packing Schemes 108

4.4.2 Efficiency of EWG 108
4.4.3 Data Analysis 109
4.4.4 Accuracy of EWG 110
4.5 Application to Parse Selection 110
4.6 Discussion 111

5 Confidence-based Training 112
5.1 Motivation 113
5.2 Research Background 114
5.2.1 Unsupervised Training 114
5.2.2 Semisupervised Training 114
5.3 Extant Parser Training and Resources 118
5.3.1 Corpora 119
5.3.2 Extant Parser Training 120
5.3.3 Evaluation 121
5.3.4 Baseline 121
5.4 Confidence-based Training Approaches 121
5.4.1 Framework 121
5.4.2 Confidence Measures 124
5.4.3 Self-training 125
5.5 Experimentation 125
5.5.1 Semisupervised Training 125
5.5.2 Unsupervised Training 130
5.6 Discussion 132

6 Conclusion 135

References 139

List of Figures

1.1 Tree and GR parser output for the sentence *The dog barked*. 14
1.2 Example sentence from Susanne. 19
1.3 Example bracketed corpus training instance from Susanne. 19
1.4 Example annotated corpus training instance from Susanne. 20
1.5 Example annotated training instance from the GDT. 20
1.6 Example sentence from the WSJ. 21
1.7 Example bracketed corpus training instance from the WSJ. 21
1.8 Example sentence from PARC 700 Dependency Bank. 22
1.9 Example of sentences from DepBank. 22

2.1 NFA for the RE $(a|b)^*ab$. 30
2.2 DFA for the RE $(a|b)^*ab$. 32
2.3 Algorithm to simulate a DFA. 32
2.4 Components of an LR parser. 33
2.5 Grammar G_1. 36
2.6 DFA for G_1. 37
2.7 LR(0) items for the rule `S` $\rightarrow$ `NP VP`. 39
2.8 Grammar G_2. 44
2.9 Grammar NFA for G_2. 45
2.10 Example parses for G_2. 46
2.11 Example graph-structured stack for G_2. 49
2.12 Example metagrammar rule. 55
2.13 The GR subsumption hierarchy 57
2.14 Simplified parse forest within the extant parser. 60
2.15 Example syntactic tree output. 62
2.16 Example n-best GR and weighted GR output. 64

3.1 RASP processing pipeline. 71
3.2 Example lexical entries in the tag dictionary. 71
3.3 Example mapping from PoS tag to terminal category. 72
3.4 PoS tag output for *We all walked up the hill* 73
3.5 SINGLE-TAG and ALL-TAG PoS tags example. 74
3.6 MULT-SYS PoS tags example. 75

4.1 The inside (e) and outside (f) regions for node N_i. 89
4.2 Calculation of inside probabilities for node N_i. 90
4.3 Calculation of outside probabilities for node N_i. 90

4.4 Example GR output using an altered GR specification. 101
4.5 Example EWG data structures for N_4. 104
4.6 Example EWG data structures for N_2. 104
4.7 Comparison of total CPU time. 108
4.8 Comparison of total memory. 109
4.9 Scatter graph of parse ambiguity to sentence length. 110

5.1 Example RASP output for a sentence from Susanne. 120
5.2 Cross entropy convergence for semisupervised EM. 128
5.3 Performance over S for C_r and EM. 129
5.4 Performance over W for C_r and EM. 129
5.5 Performance over SW for C_r and EM. 130
5.6 Tuning over the WSJ (W) from Susanne (S). 131
5.7 Cross entropy convergence for EM over unsupervised S_u. 132
5.8 Performance over S_u for C_r and EM. 133

List of Tables

1.1 Corpora summary. 24

2.1 Transition table for the NFA in Figure 2.1. 31
2.2 LALR(1) table for G_1. 37
2.3 Example stack configuration for G_1. 38
2.4 The canonical LR(0) sets for G_1. 41
2.5 Generalised LALR(1) table for G_2. 45
2.6 Example stack configurations for G_2. 48
2.7 Statistical models over the action table for G_2. 54
2.8 Example of the alternative LR parser normalisation methods. 55
2.9 Probabilities and GR specifications for nodes in Figure 2.14. 63

3.1 Tag setup descriptions. 78
3.2 Tagging Performance. 80
3.3 Parsing Performance. 82

4.1 Inside probabilities and filled GR specifications example. 98
4.2 Outside probabilities, and IO probabilities example. 99
4.3 Performance of the two parse selection algorithms. 111

5.1 Corpus split for S, W and SW. 126
5.2 Semisupervised confidence-based training performance. 127
5.3 Unsupervised confidence-based training performance. 132

Chapter 1

Introduction

This thesis develops new parse selection models and training algorithms to improve parsing accuracy and efficiency of an existing and well-developed natural language parser. This chapter first describes, in §1.1, the problem of *natural language parsing*; processing raw text to provide a linguistic analysis of the text's meaning. The work in this thesis utilises and modifies an existing well-developed parser named *RASP*. We discuss RASP's research background in §1.2, and describe the resources available for use throughout in §1.3. Finally, in §1.4 and §1.5 we describe the contributions of this thesis and provide an overview of the chapters to follow, respectively.

1.1 Natural Language Parsing

In this section, we first define the problem of *natural language parsing*. In particular, we follow previous work and define the problem as a supervised learning task. We discuss current statistical approaches to parsing and the current state-of-the-art performance for this task.

1.1.1 Problem Definition

In natural language processing, a *parser* is a computer program capable of analysing a string of words that form a well-formed sentence to determine a suitable grammatical structure: a *parse*.[1.1] Parses can be represented in a number of ways. A *parse tree* (or *syntactic tree*, tree, henceforth) represents the syntactic structure of each phrase (e.g. verb, noun or adjective phrases). *Grammatical relations* (GRs, or relational dependencies) represent the grammatical roles of different words in the sentence (e.g. subject, object). The parser's *grammar* effectively provides a mapping from word strings (i.e. terminals of the grammar) to the set of possible parses. For example, in context-free grammars (CFG) the grammar specifies syntax using simple rule rewrites to build a syntactic tree over a sentence. We use the terms analysis, derivation and parse interchangeably and consider the specific output formats as independent, given the analysis determined by the parser. For example, for the sentence: *The dog barked*, our extant parser outputs the tree and GRs in Figure 1.1.

The problem of *parse selection* is to choose the correct parse from the set of all possible parses. This task is nontrivial as large numbers of parses can result due to ambiguities in structural attachment and interaction between rules in the grammar. As a result, researchers have favoured statistical techniques whereby the *most probable parse* is assumed to be correct.

[1.1] Parsers optionally include a *sentence detection* component that marks the sentence boundaries within the raw text.

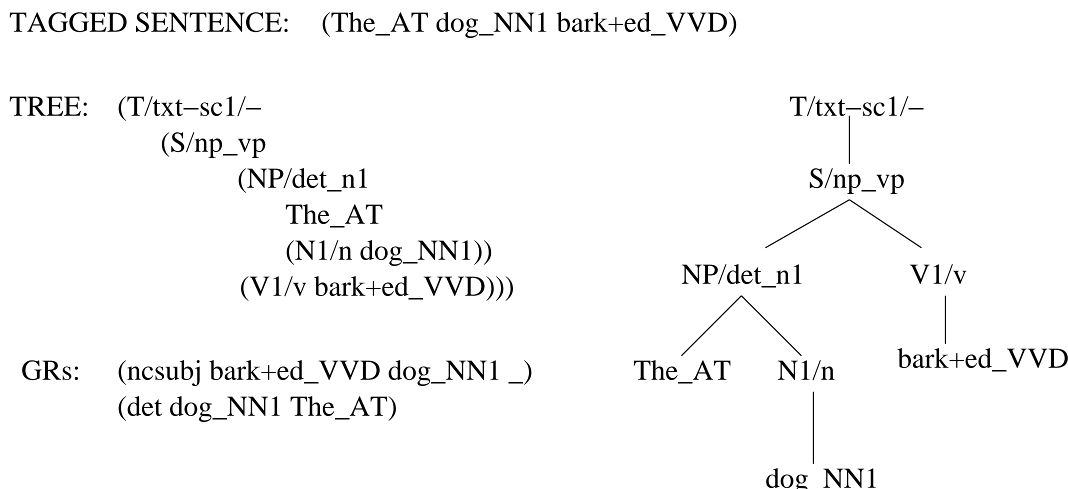

Figure 1.1: Tree and GR parser output for the sentence *The dog barked*. TAGGED is the preprocessed part-of-speech (PoS) tagged sequence for this sentence. The GR types `det` and `ncsubj` correspond to determiner and nonclausal subject relations, respectively. Note that the labelled bracketing for the tree (TREE) corresponds to the syntactic structure on the right.

Thus, parse selection is often the task of *parse ranking*, where parses are ranked from most to least probable and the first parse is considered correct. As it is infeasible to output all possible parses, parsers output only the *n* highest ranked parses (the *n*-best list).

1.1.2 Corpus-based Estimation

In order to learn the probability distributions of the statistical component of the parser, we often base the distribution on an electronic and structured text; a *corpus*. This task is often *supervised* in that the corpus, i.e. training data, will contain examples of raw text paired with the full and correct analysis. A parse ranking model should be able to model and generalise from the underlying linguistic preferences of the corpus' *domain* (e.g. newswire text). That is, a parse ranking model should model the *underlying distribution* (UD) which generates the domain's linguistic preferences, whereby the primary assumption made is that this *training data* is representative of the domain. Arguably, learning the UD over more homogeneous domains is easier than over broader domains.

The parser is applied to sentences (or raw text) termed *test data*. Ideally, both the test and training data are drawn from the same domain, whereby the training data is considered *in-domain*. In contrast, the data may instead be *out-of-domain*. Therefore, overfitting the UD is undesirable, though generalisation or inductive bias must be capable of capturing the underlying preferences manifest in this distribution.

As we aim to model the UD, training over in-domain data has been shown to outperform models trained over out-of-domain data. Gildea (2001) illustrates that training and testing Model 1 of Collins (1999) on two different corpora reduces parsing accuracy significantly. However, training over both corpora slightly increases accuracy when testing on either corpus. This illustrates the need for a balanced training corpus that can potentially generalise well to unseen domains. Even small levels of in-domain data have been shown to improve parser accuracy. For example, Tadayoshi *et al.* (2005) adapt a statistical parser trained on newswire

text to the biomedical domain by retraining on the Genia Corpus. Augmenting a parser, trained initially over a different domain, to model the preferences of another is often referred to as *parser tuning* or *domain adaptation*.

We first define parsing in terms of a supervised learning task, in which we aim to induce the function $f : X \rightarrow Y$, given training examples $\langle x_i, y_c \rangle, x_i \in X, y_c \in Y$. We define $f(x_i) \in Y$ as the selected candidate for x_i, where y_c is the correct candidate for training example i. The function f utilises the function GEN to determine set of possible candidates for x_i i.e. $GEN(x_i) \subset Y$. In parsing, $x_i \in X$ is the i-th sentence, $y_{ij} \in GEN(x_i)$ is the j-th parse for the i-th sentence and $y_c \in Y$ is the correct analysis. As y_c is often a syntactic tree, such corpora are also referred to as *treebanks*. The function $GEN(x_i)$ creates the compact data structure that represents all possible parses for sentence x_i. The size of this structure depends largely on the complexity of the particular parser's grammar.

1.1.3 Statistical Approaches

For probabilistic approaches to parsing, we utilise the supervised learning task framework, which assigns a probability to each parse $y_{ij} \in Y$ for sentence x_i. We utilise y_{ij} to denote an arbitrary parse, and y_c to denote the correct parse, for the sentence x_i. Within statistical parsers, the most probable parse is selected as the best candidate for the correct parse:

$$f(x_i) = argmax_{y_{ij} \in GEN(x_i)} Pr(x_i, y_{ij})$$

Statistical parsing models incorporate structural and/or lexical parameters (*features*) over y_{ij} that aim to include sufficient context to allow the model to learn and reflect linguistic preferences. Models differ in their choice of such features and the function *Pr*, which is used to estimate the importance of features in relation to one another. Approaches can be separated in a variety of ways. We consider *generative* or *discriminative* and *parametric* or *nonparametric* approaches. Following, we review current parsing models in terms of these types and their training criteria, and discuss the types of features they employ.

Training

Generative models define *Pr* using a joint probability model: $P(x_i, y_{ij})$. Such models are *parametric* in that they utilise a specific set of features; each with an associated weight, where the probability of a parse is based on the product of the corresponding features multiplied by their associated weights. These models are all *history-based* parsing models, as defined by Black *et al.* (1991), whereby a parse's probability is the product of the probabilities for each decision which results in creation of the parse. For example, in a probabilistic CFG (PCFG) model, the features represent the count of each CFG rule applied, while the weights represent the probability of the corresponding rule. The predominant method of training generative parametric models is to use maximum likelihood estimation (MLE) with a 'smoothing' method to allow for unseen data. MLE assigns the weights of the model so as to maximise the total probability of all parses in the training data.

In parametric discriminative models, *Pr* is the conditional likelihood $P(y_{ij}|x_i)$. That is, we model the probability only in terms of other parses for the sentence x_i. These conditional probability models (such as log-linear models) utilise iterative training schemes to ensure they maximise the log-likelihood of the training data and to compensate for dependencies amongst features. In either case, for any given sentence in the corpus, the MLE training criteria does not ensure that the likelihood of the training parse is greater than that of incorrect (competing)

parses. Nevertheless, conditional models are popular in the literature, having led to impressive ranking results when combined with large feature sets (e.g. Charniak & Johnson 2005; Clark & Curran 2004b).

In contrast, nonparametric models aim to directly differentiate between incorrect and correct parses for a given sentence. Thus all such models are discriminative and *Pr* is replaced by, for example, margin length values in maximum-margin approaches. Nonparametric approaches such as boosting, SVMs, and (variants of) the Perceptron algorithm have also yielded impressive results (e.g. Collins & Duffy 2002; Kudo *et al.* 2005; Shen & Joshi 2004). However it is unclear whether the different training criteria, or the number and type of features, provide a greater gain in accuracy. Collins & Roark (2004) report similar accuracy results for both parametric and nonparametric models over the same feature sets. Furthermore, nonparametric models are harder to train using either semisupervised or unsupervised techniques.

Models also differ in the way in which the underlying grammar is learnt. The statistical model can be learnt over a manually written grammar, or the grammar may be explicitly or implicitly learnt over a supervised corpus. Thus, to create a broad-coverage parser, manual effort is required mainly to write the grammar or create the corpus. Though some approaches, such as 'U-DOP', an unsupervised variant of the 'DOP' parsing model, assume no grammar is available and instead assume a binary branching unlabelled PCFG grammar (Bod, 2006).

Feature Spaces

PCFG models are widely acknowledged in the literature as inadequate due to their lack of context. Conversely, other parametric models such as the DOP model (Bod, 1998) should be able to model the UD accurately given explicit tree and subtree features. We can think of the level of context available (whether implicit or explicit in the model) to lie on a linear scale. At one end of the scale there are context-free models like PCFG, while at the other end we place full (sentential) context models like DOP. In general, we assume that the number of feature types and instances in a model is directly proportional to the level of context considered. Thus as we move along this scale, increasing the level of context, we decrease the level of bias, that is, the model is less likely to underfit the data. However this results in an associated increase in the number of features where the model is more likely to overfit the data; the 'bias versus variance' trade-off.

Collins (2004) highlights that due to convergence requirements and to avoid overtraining, the number of training samples should be proportional to the size of the feature space. Therefore, we should only include as many features (or as much context) as is optimal to ensure effective generalisation. Both nonparametric and conditional models are valued for their ability to incorporate a large number of (possibly dependent) features. These features are usually either n-gram statistics or CFG(-like) rules labelled with a varying amount of additional structural and or lexical context. Therefore, the latter set of features are 'DOP-like' in that they model context with features resembling subtrees (optionally labelled with lexical information).

However, the large number of lexical features (including n-gram statistics and lexicalised structural features) do not generalise well to other domains. Gildea (2001) illustrates that removing bilexical statistics (derived from the WSJ Corpus) from Model 1 of Collins (1999) decreases performance by less than 1% over the WSJ while performance was unaffected when testing over the Brown Corpus (we describe these corpora in §1.3.1). Similarly, Klein & Manning (2003) argue that lexicalised parsing models achieve around 4% absolute improvement over unlexicalised models trained and tested on the WSJ. Bikel (2004) illustrates that the bilexi-

cal parameters of Model 2 of Collins (1999) contribute less than 0.5% accuracy, while removing all of the lexical features results in a 3% decrease in accuracy when training and testing over the WSJ.

Further, efficiency is an important issue for real-world applications. All nonparametric and conditional models are reliant on an initial generative model to define the space of competing parses, the *parse forest*. Many also require the initial set of candidate parses from the parametric model to rerank (e.g. Collins & Duffy 2002; Kaplan *et al.* 2004; Kudo *et al.* 2005). Such *reranking* parsing models separate the parsing and selection phases, though some include the initial generative probability as a feature of the discriminative model e.g. Collins & Roark (2004). It is often impracticable to unpack all parses, though we would ideally like to provide the reranking model with the set of all candidate parses if this model is more accurate. Specifically, there is a trade off between efficiency and accuracy. Collins & Roark (2004) define a more efficient dynamic programming approach to enable their nonparametric model to train over and rerank all parses. However this model is only able to consider a set of local features, available at any node in the parse forest. This precludes many of the nonlocal features that can only be applied to an entire parse.

State-of-the-art Performance

Comparison of parsing systems is hampered, as performance is reported for parsers trained on different treebanks, tested over different test suites and evaluated using a number of evaluation schemes (depending on the parser's preferred output: tree or GRs). Comparison of performance over GRs is further complicated as parsers extract GRs with differing levels of granularity.

Currently, parsers considered state-of-the-art are often trained and tested on the WSJ and favour the application of highly lexicalised probabilistic models. These models report precision and recall PARSEVAL scores (Black *et al.*, 1991) of around 90% (e.g. Charniak 2000; Collins 1999; Ratnaparkhi 1999). However, the WSJ is arguably a more homogeneous and simpler corpus than other corpora, such as the Susanne treebank over which RASP is trained.

Briscoe & Carroll (2006) compare RASP's performance to that of the XLE parser of Kaplan *et al.* (2004) and of Model 3 of Collins (1999). Though the comparison is nontrivial, they illustrate that RASP is substantially faster than both parsers, with parser accuracy higher than that of the Collins' parser and ranging between 'cut-down-grammar' and 'complete-grammar' XLE systems. Further RASP is trained over a subset of the Brown Corpus, that is, an out-of-domain training set. Therefore, RASP's accuracy and efficiency is arguably state-of-the-art.

Discussion

Nonparametric models which aim to directly differentiate correct from incorrect parses are complex to train and often inefficient to decode. Nevertheless, they are currently popular because of their ability to accurately model the UD and ensure that the probability (or score) of correct parses exceeds that of competing incorrect ones for training inputs. Other recent parametric models, such as log-linear models, are also complex to train and inefficient to decode. However, if compared to nonparametric models using the same feature sets and training data, they produce similar ranking accuracy (e.g. Collins & Roark 2004).

Generative parametric models capable of direct MLE are currently disfavoured because empirically their performance has been worse. Briscoe & Carroll (1993) demonstrate that a generative probability distribution defined over the actions of a (nondeterministic) generalised LR parser (GLR, Tomita 1987) provides additional context to the ranking model obtained from a standard PCFG. Further, the parser retains advantages such as simple estimation and efficient

decoding. In this thesis we utilise this GLR parser, in search of parametric models that can be easily trained and efficiently decoded and which readily support semisupervised training techniques.

1.2 Research Background

RASP resulted as part of the tool-set developed within the Alvey Natural Language Tools (ANLT) for English at the University of Cambridge funded by the UK Alvey Programme. The parser is a generative (parametric) model based on the GLR framework. [1.2]

RASP (the 'robust accurate statistical parser') is a state-of-the-art system for text processing distributed freely for research purposes.[1.3] It is a set of pipelined modules for sentence boundary detection, word tokenisation, part-of-speech tagging and syntactic parsing which recovers the grammatical relations between words, phrases and clauses in individual text sentences. The full (Lisp and C) code base runs to several hundred thousand lines. The RASP system, described by Briscoe & Carroll (2002), has been downloaded by over 160 groups. This paper, describing the first release, has been cited 147 times (Google Scholar 14/08/07) in descriptions of further research utilising RASP by researchers in the UK, Europe, the US, Australia and Asia. It has been used to automatically annotate over 1 billion words of English text in the context of published work developing lexical databases, question-answering systems, text classifiers, information extraction systems, summarisers, and so forth. The second release of this system is described by Briscoe *et al.* (2006).

1.3 Available Resources

This thesis is the result of research conducted at the Computer Laboratory of the University of Cambridge, under the supervision of Professor E.J. Briscoe, who co-developed RASP. As a result, this work benefits from direct access to RASP's grammar, code and all associated processing components. RASP is utilised in a number of research projects both within the Computer Laboratory, and in other universities and in commercial projects. As a result the grammar, associated processing components and also evaluation schemes were developed, and thence, varied throughout the life of this work. The data and evaluation schemes used throughout this work are described in the following sections. Results in subsequent experimental chapters are comparable within their respective chapters only due to changes that occurred in the grammar and/or evaluation framework.

1.3.1 Corpora

The following sections describe data in use throughout this work as training, tuning or test data. The treebanks we use in this work are in one of two possible formats. In either case, a treebank T consists of a set of training instances. Each training instance $t \in T$ is a pair (s, M), where s is the automatically preprocessed sentence text (tokenised and labelled with PoS tags, see §3.2) and M is either a fully annotated derivation, A, or an unlabelled bracketing U. This bracketing may be partial in the sense that it may be compatible with more than one derivation produced by a given parser. Although occasionally the bracketing is itself complete, the alternative nonterminal labelling causes indeterminacy. Often the 'flatter' bracketing available from existing treebanks is compatible with several alternative 'deeper' mostly binary-branching derivations output by a

[1.2] We provide details of this framework, and specific details of the RASP system, in the following chapters.

[1.3] See http://www.informatics.susx.ac.uk/research/nlp/rasp/ for license and download details.

parser.

Susanne Treebank

The Susanne Corpus (henceforth, Susanne) is a balanced subset of the Brown Corpus which consists of 15 broad categories of American English texts (Sampson, 1995). This treebank contains detailed syntactic derivations represented as trees, but the node labelling is incompatible with our system's grammar. Figure 1.2 illustrates an example sentence from the corpus. The RASP developers built two system-compatible corpora from Susanne, a bracketed corpus and an annotated corpus.

To build the bracketed corpus, sentences were extracted from Susanne and automatically preprocessed. A few multiwords were retokenised, and the sentences were retagged using RASP's PoS tagger. The bracketing was then automatically and deterministically modified to more closely match that of RASP's grammar. This pipeline resulted in a bracketed corpus of 7014 sentences. Figure 1.3 illustrates the corresponding bracketed corpus training instance extracted from the original annotation in Figure 1.2. The first item (line) is the preprocessed word-stems with the corresponding PoS tags (determined using the original word rather than the stem). The second line provides unlabelled bracketing over the words of the sentence.

```
A01:0700a    -    APPGm   His        his        [S[Ns:s.
A01:0700b    -    NN1c    petition   petition   .Ns:s]
A01:0700c    -    VVDv    charged    charge     [Vd.Vd]
A01:0700d    -    JJ      mental     mental     [Ns:o.
A01:0700e    -    NN1n    cruelty    cruelty    .Ns:o]S]
A01:0700f    -    YF      +.         -          .O]
```

Figure 1.2: Example sentence from Susanne. The third column illustrates the PoS tag while the fourth and fifth column show the word and word-stem, respectively. The final column illustrates the syntactic structure of the sentence.

```
his_APP$ petition_NN1 charge_VVN mental_JJ cruelty_NN1 .
((his petition) charge (mental cruelty))
```

Figure 1.3: Example bracketed corpus training instance from Susanne.

A fully annotated and system compatible treebank of 4801 training instances (3543 of which are unique) from this bracketed corpus was also created. The system parser was applied to construct a parse forest of analyses which are compatible with the bracketing. For 1258 training instances, the grammar writer interactively selected correct (sub)analyses within this set until a single analysis remained. The remaining 2285 training instances were automatically parsed and all consistent derivations were returned. Since the bracketing is consistent with more than one possible derivation for roughly two thirds of the data, the 1258 training instances were repeated twice so that counts from these trees were weighted more highly. The level of reweighting was determined experimentally using some held out data from Susanne. Even given the partial

```
(his_APP$ petition_NN1 charge_VVN mental_JJ cruelty_NN1 ._.)
(T/txt-sc1/-+
 (S/np_vp (NP/det_n1 his_APP$ (N1/n petition_NN1))
  (V1/v_n1 charge_VVN
   (N1/ap_n1/- (AP/a1 (A1/a mental_JJ)) (N1/n cruelty_NN1))))
 (End-punct3/- ._.))
```

Figure 1.4: Example annotated corpus training instance from Susanne.

bracketing derived from Susanne, the costs of deriving the fully annotated treebank are high, as interactive manual disambiguation takes an average of ten minutes per sentence.

Returning to our previous example, Figure 1.4 illustrates the fully annotated training instance. Again, the first line contains the preprocessed text while the second element (subsequent lines) correspond to the fully annotated derivation that will be directly compatible with (a specific version of) RASP's grammar.

Grammar Development Treebank

The grammar development treebank (GDT) is an annotated corpus, a list of around two thousand manually maintained sentences paired with correct derivations. Figure 1.5 illustrates an example annotated training instance from the GDT. Each training instance consists of a pair; the preprocessed text and a corresponding derivation which is compatible with the current version of RASP's grammar. The GDT does not include useful frequency information as, in general, each grammatical rule occurs in derivations as many times as is required to illustrate its interaction with other rules to define the linguistic coverage of the system. Furthermore, these sentences are short and relatively artificial, having been constructed by the grammar writer to elucidate coverage and minimise ambiguity.

```
(The_AT technology_NN1 is_VBZ long_JJ in_II the_AT tooth_NN1)
(T/txt-sc1/--
 (S/np_vp (NP/det_n1 The_AT (N1/n technology_NN1))
  (V1/be_ap/- is_VBZ
   (AP/a1
    (A1/a_pp long_JJ
     (PP/p1
      (P1/p_np in_II
       (NP/det_n1 the_AT (N1/n tooth_NN1)))))))))
```

Figure 1.5: Example annotated training instance from the GDT.

Wall Street Journal

The Penn Treebank (PTB) is a corpus of over 4.5 million words of American English (Marcus *et al.*, 1993). The corpus has been annotated with PoS tags and around half has been annotated with skeletal syntactic structure. As this corpus is the largest treebank available for English,

```
((S
   (NP-SBJ (DT The) (JJ new) (NN rate))
   (VP (MD will)
     (VP (VB be)
       (ADJP-PRD (JJ payable)
         (NP-TMP (NNP Feb.) (CD 15)))))
   (. .)))
```

Figure 1.6: Example sentence from section 2 of the WSJ. Brackets are labelled with the phrasal category or with the PoS tag for each word.

```
The_AT new_JJ rate_NN1 will_VM be_VB0 payable_JJ Feb._NPM1 15_MC ._.
((The new rate) (will (be (payable (Feb. 15)))) .)
```

Figure 1.7: Example bracketed corpus training instance from the WSJ.

the Wall Street Journal (WSJ) sections of the Penn Treebank (PTB) are employed as both training and test data by many researchers in the field of statistical parsing. The annotated corpus implicitly defines a grammar by providing labelled bracketing over words annotated with PoS tags. An example annotated sentence is shown in Figure 1.6.

We extract the unlabelled bracketing from all sections of the WSJ, including those for the de facto standard training sections (2-21 inclusive). The pipeline is the same as that used for creating the bracketed Susanne corpus. However we do not automatically map the bracketing to be more consistent with the system grammar. Instead we simply remove unary brackets. The de facto training set is compiled to form a bracketed corpus of 38,329 training instances. Figure 1.7 shows the resulting bracketed treebank training instance for the corresponding WSJ sentence shown in Figure 1.6.

Parc 700 Dependency Bank

King *et al.* (2003) describe the development of the PARC 700 Dependency Bank, a gold-standard set of relational dependencies for 700 sentences drawn at random from section 23 of the WSJ (the de facto standard test set for statistical parsing). Briscoe & Carroll (2006) parse the corpus with RASP (and manually correct the output if required), to create a doubly annotated data set, enabling (nontrivial) comparison of RASP with other state-of-the-art statistical parsers. Figure 1.8 illustrates an example sentence from the resulting corpus with both parser's annotation.

We test our parser on the same 560 sentence subset (DepBank, henceforth) that Kaplan *et al.* (2004) utilise in their study of parser accuracy and efficiency. Unless otherwise stated we utilise DepBank without gold standard PoS tagging, that is, we apply our entire automated pipeline including PoS tagger. A gold standard named-entity (NE) mark-up for DepBank was provided by Stephan Riezler, co-author of Kaplan *et al.* (2004). The gold standard RASP annotation over the NE markup was also developed in the format shown in Figure 1.8. Thus there are two gold-standards for DepBank, one with and one without NE mark-up.

As this is a test corpus, we build the corresponding files to parse that consist only of the

```
sentence(
  id(wsj_2351.19, parc_23.15)
  date(2002.6.12)
  validators(T.H. King, J.-P. Marcotte)
sentence_form(But that won't be easy.)
structure(adjunct(be~0, but~6)
  adjunct(be~0, not~5)
  stmt_type(be~0, declarative)
  subj(be~0, pro~2)
  tense(be~0, fut)
  xcomp(be~0, easy~1)
  adegree(easy~1, positive)
  subj(easy~1, pro~2)
  num(pro~2, sg)
  pron_form(pro~2, that)
  adegree(but~6, positive))
rasp(
(conj But be)
(ncsubj be that _)
(aux be wo)
(ncmod _ easy n't)
(xcomp _ be easy)))
```

Figure 1.8: Example sentence from PARC 700 Dependency Bank annotated using both RASP and the XLE parser of Kaplan *et al.* (2004).

```
But_CCB that_DD1 wo_VM n't_XX be_VB0 easy_JJ ._.
Would_VM <w>Mr. Antori_NP2</w> ever_RR get_VV0 back_RL in_II ?_?
```

Figure 1.9: Example of sentences from DepBank.

preprocessed sentences. To do so, we extract the raw text from the NE and non-NE DepBank corpora then process this text (e.g. tokenise and tag) using RASP's preprocessing modules. The first line of Figure 1.9 shows an example preprocessed sentence, which corresponds to the sentence in Figure 1.8. The second line illustrates an example sentence with NE mark-up.

Kaplan *et al.* (2004) report results in terms of both relational dependencies, which are the reformatted and corrected F-structure output from their lexical functional grammar (LFG) parser, and also a set of 'semantically-relevant' features. In our modified version of DepBank, we replace these features with a slightly richer relational annotation (Briscoe & Carroll, 2006). Briscoe & Carroll (2006) note that RASP's parsing results (F_1 scores) are lower than those reported in that paper partly because many of these features are numerous and relatively easy to recover.

Summary

Susanne is a (balanced) subset of the Brown Corpus which consists of 15 broad categories of American English texts. All but one category (reportage text) are drawn from different domains than that of the WSJ. Therefore we consider Susanne as out-of-domain training data when testing on DepBank, following Gildea (2001) and others.

We provide a summary of the data in Table 1.1. Note that the average parses per sentence and average GRs per sentence will differ depending on the version of the grammar used[1.4] to parse the data. However, these statistics provide a basis for comparison of complexity. The default system processing limitations were imposed on our parser (see §2.6.3) for all corpora except DepBank, halting the parser for some sentences prior to completing parsing. Further, we considered only sentences with a word length of less than or equal to 50. Thus the number of sentences (Sent) in the corpus is shown, as well as the number of sentences that completed parsing (Comp). From this latter set we determined the statistics for sentence parses and GRs. The final column (Frag) illustrates the number of parses for which a *fragmentary parse* (see §2.6.4) was found. That is, the system was unable to find a full parse given the grammar, though was still able to return an analysis for the sentence.

1.3.2 Evaluation

We evaluate the parser's output using a relational dependency evaluation scheme (Carroll *et al.*, 1998; Lin, 1998) with standard measures: precision, recall and F_1. Relations are organised in a subsumption based hierarchy. In addition to determining precision, recall and F_1 for each level in the relation hierarchy, we calculate the micro- and macro-averaged values for each of these scores across all relations. The macro-average measure is calculated by taking the average of each measure for each individual relation, while the micro-average measure is calculated from the counts across all relations.

As previously mentioned, many parsers currently report PARSEVAL evaluation measures based on the labelled bracketing of the parser. That is, a bracket is labelled with a nonterminal grammar category and inside the bracket are other nonterminal categories (more labelled brackets) or words. This 'bracketing' provides a flat representation of a syntactic tree. Therefore this evaluation compares the structure of a parse tree output by the parser with that of the gold standard tree. In contrast, the relation-based evaluation considers the GRs output from the parser, where the same GR can be produced within different syntactic structures. As a result, our evaluation does not penalise different structural representations but instead aims to provide merit to the semantics extracted.

Hierarchy Based Evaluation

For each sentence we determine the relations output by the parser that are correct at each level of the relational hierarchy. This hierarchy is shown in Figure 2.13, in a subsequent section in which we describe RASP's grammar. Relations take the general form: `(relation subtype head dependent initial)`. A relation is correct if the head and dependent slots are equal and if the other slots are equal (if specified). The evaluation we perform in the final experimentation chapter (Chapter 5) differs in two respects from evaluation defined in Carroll *et al.* (1998). Firstly, a different relational hierarchy is applied due to a major change in grammar (described in Briscoe & Carroll 2006). Secondly, the evaluation scheme is altered so that GR counts

[1.4]Those quoted here were determined over the *tsg15* grammar.

Data	Sent	Comp	Words				Parses				GRs				Frag
			range	avg	stdev	med	range	avg	stdev	med	range	avg	stdev	med	
Susanne	7014	6597	2–94	19.52	10.87	18	1–3.28e9	1.44e6	4.63e7	55	0–82	16.16	10.03	15	850
GDT	2079	2079	2–22	6.60	2.40	6	1–356	4.12	12.56	2	0–18	4.93	2.04	5	0
WSJ	39617	37094	1–141	23.98	11.41	87	1–1.04e9	7.95e5	1.47e7	146K	0–49	18.57	8.72	43	6876
DepBank	560	560	3–61	22.82	10.34	22	1–5.95e10	1.07e10	2.51e11	148	1–53	18.84	9.25	18	110

Table 1.1: Summary of corpora used throughout this work. The WSJ statistics are calculated over the de facto training sections 2-21 inclusive.

are percolated upwards throughout the hierarchy. This enables the root GR type `dependent` to represent the unlabelled dependency scores.[1.5]

In the new evaluation, if a relation is incorrect at a given level in the hierarchy it may still match for a subsuming relation (if the remaining slots all match). For example, if an `ncmod` relation is mislabelled with `xmod`, it is correct for all relations which subsume both `ncmod` and `xmod`, e.g. `mod`. Similarly, the GR is considered incorrect for `xmod` and all relations that subsume `xmod` but not `ncmod`. Thus, the evaluation scheme calculates unlabelled dependency accuracy at the `dependency` (most general) level in the hierarchy. The micro- and macro-averaged precision, recall and F_1 scores are calculated as they were in the previous evaluation; from the counts for relations in the hierarchy.

Wilcoxon Signed Ranks Test

In statistics, a result is *significant* if it was unlikely to have occurred by chance. A *statistically significant* result means there is statistical evidence that there is a difference (not necessarily large) between two sets of data. We can utilise statistical significance to compare two parsers, by comparing whether their performance was statistically significant. Therefore, we can determine whether a change in the parsing model improves the parser's accuracy or if an increase in accuracy occurs simply by chance.

The Wilcoxon Signed Ranks (Wilcoxon, henceforth) test is a *nonparametric* test for statistical significance that is appropriate when there is one data sample and several measures. For example, to compare the accuracy of two parsers over the same data set. As the number of samples (sentences) is large we use the normal approximation for z. Siegel & Castellan (1988) describe and motivate this test. These results are computed over micro-averaged F_1 scores for each sentence in the test corpus.

We first determine C_d; the subset of sentences in the test corpus C for which the accuracy differs between parsers. We determine the set of accuracy differences between the parsers for each sentence in C_d, ranking the (absolute value) of these differences in order from smallest to largest in a list D. The statistic T_+ is the sum of the ranks of the positive, non-zero differences, where $d_i \in D$ is the difference ranked i (this ranking starts at 1):

$$T_+ = \sum_{d_i \in D, d_i > 0} i$$

Mean, variance and observed z are determined as follows, where N is the size of the set C_d:

$$Mean = \mu_{T^+} = \frac{N(N+1)}{4}$$

$$Variance = \sigma^2_{T+} = \frac{N(N+1)(2N+1)}{24}$$

$$z = \frac{T^+ - \mu_{T^+}}{\sigma_{T+}}$$

We use a 0.05 level of significance, which indicates that there is a 5% chance that we incorrectly find the results are statistically significant when they are not. We provide z-value probabilities for significant results reported below, where a result is statistically significant if

[1.5] We define the new grammar and GR hierarchy only. Readers are referred to Briscoe *et al.* (2002) for a diagram of the previous GR hierarchy.

the z-value probability is less than 0.05. Therefore, providing the z-value probabilities enables the reader to ascertain the level of statistical significance that applies. For example, a z-value probability of 0.01 is judged significant in this work (as it is less than 0.05), and indicates that there is a 1% chance that this conclusion is incorrect.

1.4 Research Goals

The work in this thesis aims to improve parser accuracy, efficiency and training methods of the extant parser. While these methods utilise (and modify) RASP's extant parsing code and components, the parse selection and training methods we develop herein are applicable to a wider range of statistical parsers.

Initial experimentation, including optimising PoS tag models and developing an efficient algorithm to determine the weighted GR output format, aids in the author's familiarisation of the RASP system and processing modules. Further experimentation predominantly aims to improve the training methods available to the parser. In particular to develop semisupervised training methods, which require no on-going manual effort on behalf of the grammar writer, to facilitate grammar development.

1.5 Thesis Summary

1.5.1 Contributions of this Thesis

The research in this thesis provides a number of new results and techniques (all techniques and the majority of results have been published), in the following areas:

- *Optimising front-end PoS tagging models:* Watson (2006) provides a broad comparison of PoS tag models in terms of both tagging and parsing performance. Experimental results illustrate that parsers are unable to improve on the tagging performance of a 'good' PoS tagger. Resolving the majority of PoS tag ambiguity in the tagger aids in both parser accuracy and the overall system's efficiency.

- *Optimising weighted-GR output:* Watson *et al.* (2005) define a novel modification to the Inside-Outside algorithm. This efficient dynamic programming approach directly determines the weighted-GR output format from the compact representation of parses; the parse forest. The approach improves over previous work, which either loses efficiency by unpacking the parse forest, or places extra constraints on local ambiguity packing, leading to less compact forests. This novel algorithm significantly increases the throughput and accuracy of this output format.

- *Semisupervised parser training:* Watson *et al.* (2007) illustrate more efficient and flexible use of existing training data. The Inside-Outside algorithm is applied to perform Expectation-Maximisation over the LR parse table to improve performance over the current, fully supervised training method. Removing the manual effort required to train the parser facilitates grammar development and domain adaptation.

- *Confidence-based semisupervised training:* Watson *et al.* (2007) describe a new training framework that defines a weighting scheme over analyses considered consistent during training. This method outperforms the Inside-Outside algorithm given the same level of corpus annotation. Further, the method can be applied to any level and type of annotation

by simply redefining which parses are compatible with the corpus annotation available during training.

1.5.2 Outline of Subsequent Chapters

Chapter 2 (*LR Parsers*) describes the theory, compilation, and application of LR parsers or so-called shift-reduce parsers. We describe the modifications defined by Tomita (1987) which extend the LR framework to ambiguous grammars resulting in the GLR parsing framework. We then describe various methods for defining statistical distributions over the GLR model. That is, to create statistical GLR (SGLR) parsers. Further, the chapter defines the SGLR parsing framework employed within RASP. Finally, we describe RASP's grammar, extant training and application of the parser as well as the different output formats available for use.

Chapter 3 (*Part-of-speech tag models*) describes experimentation aimed at optimising the PoS tag models employed by the extant parser. We first describe previous work which illustrates that the choice of the PoS tag model employed as a front-end to parsing significantly affects the speed and accuracy of the parser. Next, we describe RASP's processing components, in particular, the current PoS tagger. We define a number of different tag selection models, including those that consider the parser itself as a PoS tagger. Finally, we investigate the optimum level of PoS tag ambiguity to be resolved by the parser itself, rather than the front-end PoS tagger.

Chapter 4 (*Efficient extraction of weighted GRs*) first describes the Inside-Outside algorithm (IOA) and its application to train PCFGs. We extend this algorithm to SGLR parsers, in particular, to the extant parsing framework. We show how to apply a variant of the IOA to efficiently extract one of RASP's output formats ('weighted GRs') directly from the parse forest rather than over the set of n-best parses (the extant method). Furthermore, we illustrate that this solution can not be applied in all situations. Rather than modify the extant parse forest as previous work has, resulting in a less compact data structure, we describe a novel modification to the IOA. This modification enables *multiple* inside and outside probabilities to be determined for each node within the parse forest. Experimental results compare this novel approach to the current one, in terms of both parser throughput and accuracy.

Chapter 5 (*Confidence-based training*) reviews the current training method and describes the limitations of such fully supervised training approaches. These limitations have prompted the development of unsupervised and semisupervised methods which we describe. We then define a confidence-based training framework which can be applied over any level or type of corpus annotation, and its relationship to previous work. We also define a number of different confidence measures that can be employed within this framework. Experimental results compare these confidence measures and the IOA (Expectation-Maximisation), over semisupervised and unsupervised training corpora, to the current training method.

Chapter 6 (*Conclusion*) summarises the major contributions of this thesis, and suggests future lines of research.

Chapter 2

LR Parsers

This chapter describes the theory and application of *LR parsers* which are a type of *shift-reduce* parser. We define LR parsers and generalised LR (GLR) parsers in §2.3 and §2.4, respectively. These descriptions build on the theory of finite-state automata we provide in §2.2. We describe existing statistical approaches over the latter parsing framework in §2.5. Experiments in this work use an existing and well-developed GLR parsing system which we modify as required. Finally, we provide details specific to the extant parser in §2.6. Much of the theory described herein regarding finite automata and LR parsing (including parser compilation) has been adapted from Aho *et al.* (1986).

2.1 Introduction

The LR parsing strategy was first devised for programming languages, in particular compilers, to enable precompilation of processing steps over a language. The strategy has since been generalised for use in a wider range of applications, including natural language parsing. We previously defined a *natural language parser* (in §1.1.1) as a program analysing a string of words (sentence) to return a suitable grammatical structure: a *parse*. However, in the literature, the term *parser* refers to any program that processes a sequence of input tokens to return structure over these tokens, given an underlying *formal language* or *grammar*. In linguistics, the term *grammar* applies to various structures of human language including phonology, morphology and syntax.

There are many types of grammars. We describe context-free grammars (CFG), a well known type of generative grammar. A CFG defines the set of possible input tokens, the grammar *terminals* (that is, the alphabet Σ) of the grammar, in the *lexicon*. The *production rules* (*rules* or *productions* for short) define the way in which these terminals may acceptably combine. The primary goal of a parser is to organise the input sequence of terminals, based on the rules, into larger grammatical units or *phrases*. These larger units are referred to as *nonterminal* (NT) symbols of the grammar. Rules are written as *rewrites*: $A \rightarrow (\beta)^*$ where A represents a NT symbol of the grammar and β may be any *category* of the grammar i.e. a Σ or NT symbol. For example, the rule `NP` $\rightarrow$ `Det N` stipulates that a noun phrase `NP` may be formed by two terminal symbols; a determiner (`Det`) followed by a noun (`N`).

As the NT categories combine Σ categories of the grammar, we consider this set of terminals the *span* (or *word span*, given that terminals are often words in natural language parsing) of the resulting NT category. One (or more) of these NT symbols are considered *root* (or *top*) categories. If such a top category spans the entire input sequence, then we accept the input and

return the corresponding structure; a *parse* (or *derivation*). This structure can be graphically represented as a tree diagram (see Figure 1.1) where leaves are terminals while nonleaf nodes represent nonterminals (phrases) of the grammar. Hence, the term "parse tree" (or "phrase-structure tree") is used to refer to this structure.

We can formally define a grammar G using the tuple $\{NT, \Sigma, P, R\}$ where:

- NT is the finite set of nonterminal symbols.
- Σ is the finite set of terminal symbols, disjoint from the set NT.
- P is the finite set of production rules of the grammar:

$$NT \rightarrow (\Sigma|NT)^*$$

- $R \in NT$, the root category symbol. [2.1]

We first define *finite-state automata* (FSA), that can be described with regular expressions (RE). REs are one way of characterising regular languages (a proper subset of the languages generated by a CFG). We describe how FSAs are graphically represented as *transition graphs*, though are encoded and processed by the parsing program using the corresponding *transition table*. We then describe *LR* parsers, that implicitly encode probabilistic deterministic FSA (PDFA), thence also utilise a similar table, the *LR table*, to drive the parser.

2.2 Finite Automata

If the parser's grammar identifies acceptable sequences of terminals only, without associating structure, then the parser is instead considered a *recogniser*. A recogniser takes as input a sequence of tokens and returns either 'accept' or 'reject' as output. For example, given the RE $(a|b)^*ab$, we only accept a sequence of input tokens if it consists of any number of a and b tokens (including 0) followed by the sequence ab. The sequences ab and $bbabab$ are accepted, while $aabb$ is not.

FSA can apply either as a parser or as a recogniser given a language (grammar). Aho *et al.* (1986) describe a number of algorithms to compile an FSA from a RE, which we do not discuss here. We aim instead to illustrate how these automata are used as recognisers for RE, and build on this theory in subsequent sections to illustrate the automata implicit in LR parsers over CFGs.

2.2.1 NFA

Model Definition

A nondeterministic FSA (NFA) is defined using:

- A set of states S.
- A set of input symbols T.
- A transition function $action$ that maps state-symbol pairs to the set of next possible states.

[2.1] While a single root category symbol is defined, the grammar in the existing parser RASP defines several root category symbols. Thus, the following explanations assume more than one root category symbol may be defined in the parser's grammar.

- A state $S_0 \in S$, which is the *start* (or *initial*) state.
- A set of states $S_{acc} \subseteq S$, the set of *end* (or *accept*) states.

A FSA is deterministic or nondeterministic, where *nondeterministic* means that more than one transition out of a state is possible on the same input symbol. That is, *action* returns one or more possible states for any given state-symbol pair. The null input symbol ε is included in T.

Transition Graph

The *transition graph* diagrammatically represents an NFA as a labelled directed graph. Vertexes of the graph each correspond to a state in the set S, while labelled edges represent the output of the function *action*. The graph looks like a transition diagram, where edges are labelled with terminals of the grammar i.e. $T = \Sigma$. Given the RE in the previous example, $(a|b)^*ab$, the corresponding transition graph is shown in Figure 2.1. In this example, the NFA is defined over $S = \{0,1,2\}$ where $S_0 = 0$ and $S_{acc} = 2$[2.2], $T = \{a,b\}$, and *action* is implicit in the labelled edges of the graph. For example, $action(1,b) = \{2\}$ while $action(1,a) = \{\}$.

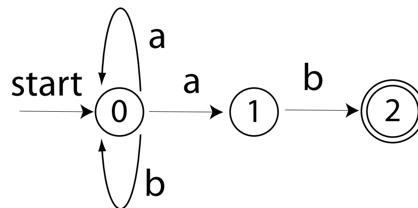

Figure 2.1: NFA for the RE $(a|b)^*ab$.

Paths

An NFA acting as a recogniser, accepts a sequence of input symbols if we can find a *complete path* through the NFA from the start state (S_0) to an end state (in S_{acc}) which consumes the entire input sequence. A *path* represents a *state sequence*, the set of states visited thus far given the input consumed. For example, we accept the input sequence *abab* as we determine the complete path $\{0,0,0,1,2\}$ through the NFA.

To determine a path we utilise two variables: the current state S_c and the remaining input terminals. To start, we initialise these variables to the start state $S_c = S_0$ and entire input sequence, respectively. The transitions defined by *action* (edges) are *parsing actions*, these movements are conditioned on the current state S_c and next token in the input sequence. When we move along each edge of the NFA, we consume the terminal in the input sequence that labels the edge. Returning to our previous example, for the sequence *abab*, in state 1 we have consumed *aba* and have the remaining input $\{b\}$. Thus we move from state 1 to state 2 consuming b and, reaching one of the end states with no remaining input, we return 'accept'.

Transition Table

A *transition table* is used as an operational mechanism to drive the parser across the corresponding transition graph. There is a simple mapping between row and columns in the table to states and terminals (edge labels) in the graph. Thus the table can be determined from the graph

[2.2] We utilise the state number i in S_i to identify each state.

and vice versa. For example, Table 2.1 illustrates the corresponding transition table for the NFA graph in Figure 2.1. Each row in the transition table represents a state in the NFA, while columns each represent a terminal symbol. Within each cell in the table, in row i and column j, is the set of possible edges (actions) of the NFA for state S_i and the j-th terminal (Σ_j). Thus, the table encodes the function *action*, where each cell contains the set of states returned by *action* given the state-symbol pair.

STATE	INPUT SYMBOL	
	a	b
0	0,1	0
1		2
2		

Table 2.1: Transition table for the NFA in Figure 2.1.

If *action* returns null (i.e. for empty cells in the table) this implies an edge to the *sink state* exists, at which point the input sequence is rejected. As a result, we consider the corresponding path to *terminate* at the current state. A path is considered *active* until it either terminates or completes.

Multiple Paths

More than one path may be possible through an NFA given an input sequence. Multiple entries in a single cell represent such ambiguous transitions. In this case, we continue processing all active paths until we determine a single complete path, at which point we accept the input. Alternatively, if all active paths terminate, we reject the input. At the state where multiple paths (edges) are found, we consider the paths to *diverge*. Conversely, paths are considered to *merge* for shared portions of their state sequence.

For the example sequence *abab* if we consume the first terminal *a* then two paths are active; one ending each in state 0 and state 1. From these states, we consume *b* and move to states 0 and 2, respectively. At state 2, our path terminates as there is still the unexpended input $\{a,b\}$. In contrast, the path ending at state 0 remains active, though again we diverge to two paths: to states 0 and 1. After consuming the last token *b*, the first path terminates and the second path ends at state 2. The latter path ends at an accept state and is now considered complete.

Formal Language

An NFA implicitly defines its *formal language*; the set of all input strings it accepts. Returning to our previous example, the formal language is the set: $\{ab, aab, bab, aaab, baab, abab, bbab, \ldots\}$.

2.2.2 DFA

A deterministic FSA (DFA) is a special case of an NFA in which only one move is possible for any given state-symbol pair, and no state has an edge with the null symbol ε. Aho *et al.* (1986) describe an algorithm to compile an NFA to its equivalent DFA, that results in a more efficient recogniser. However, the number of states of the DFA may expand exponentially and this conversion is infeasible for large NFA. We do not provide details of this algorithm as they are not relevant to this work. Figure 2.2 illustrates the equivalent DFA for the NFA in Figure 2.1.

Given the automaton is deterministic, we utilise a list to store the corresponding state sequence (path) and input consumed. This list is referred to as the *stack*. The stack U is initially

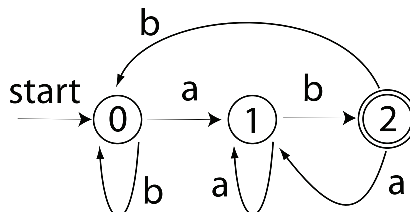

Figure 2.2: DFA for the RE $(a|b)^*ab$.

empty, and new states and input symbols consumed are added to the stack as we move through the NFA. Continuing the previous example, the sequence *abab* was accepted with state sequence $\{0,0,0,1,2\}$. This state sequence is determined given the stack built during parsing: $\{0,a,0,b,0,a,1,b,2\}$.

Algorithm: Simulating a DFA

The algorithm in Figure 2.3 simulates a DFA, taking as input a sequence terminated by the end-of-file character (eof) and returning either 'accept' or 'reject'. The function *next* returns the next terminal in the input sequence.

Algorithm 1 (Simulating a DFA)**.**
Set $S_c = S_0$ *and* $U = \{S_0\}$

1. repeat forever begin
  2. $l = next()$
  3. Set $S_c = action(S_c, l)$
  4. if $S_c \in S$ *then* $U = U \cup \{l, S_c\}$*;*
  5. if $S_c \in S_{acc}$ *and* $l = eof$ *then return 'accept';*
  6. if $S_c = null$ *or* $l = eof$ *then return 'reject';*
end

Figure 2.3: Algorithm to simulate a DFA.

2.3 LR Parsers

LR parsing is a table-driven technique, where the *LR table* is constructed from an unambiguous CFG and implicitly encodes a DFA. The LR table defines the next parsing action (edge of the DFA), so that the parser is driven over a sequence of input terminals in a similar fashion as described for a recogniser over a DFA. Consequently, we move over the input left-to-right, hence the 'L' in 'LR' parsing. We reduce the input consumed to form non-terminal categories whenever possible which results in creating the 'right-most derivation in reverse', hence the 'R' in 'LR'.

The LR precompilation process results in construction of the LR table over the CFG, similar to the transition table described previously. This process represents the bulk of the processing required, and subsequently, the parsing process is relatively simple to encode and apply. We describe this precompilation process in detail in this section. Algorithms that generalise these

methods to apply over ambiguous grammars, and moreover, to define statistical models within this generalised framework, are discussed in subsequent sections.

2.3.1 LR Parsing Model

In this section we describe how to compile and apply an LR parser over CFGs, in which rules are rewrites as defined previously: $NT \rightarrow (\Sigma|NT)^*$. Following popular terminology, we refer to the left hand side category as the *mother* category, while terminals and nonterminals on the right hand side of the rule are *daughters* of the production.

Relationship to the Recogniser

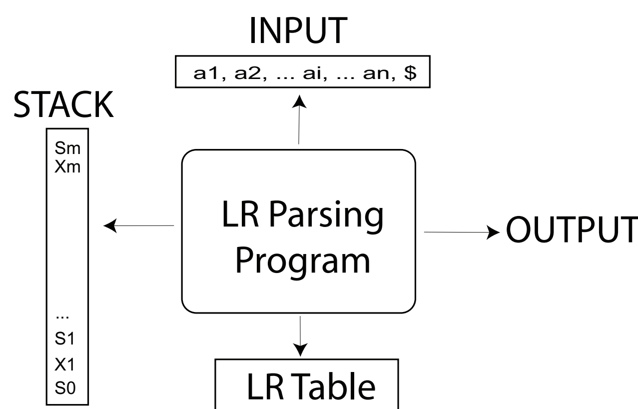

Figure 2.4: Components of an LR parser.

Figure 2.4 illustrates the general components of an LR parser, similar to those of the recogniser described in the previous section. In summary, the LR parser we describe differs from the recogniser as follows:

- *Input:* again consists of sequences of terminals of the grammar (from the set Σ). Although Σ does not include ε and instead includes the end of sentence marker $. Input sequences are appended with the symbol $ (rather than eof) to denote the end of the input sequence.

- *Parsing actions:* consist of the type of actions described previously (consuming a terminal symbol, accepting and rejecting input) as well as a fourth action type: *reduce* actions, in which rules (rewrites) of the regular grammar are applied to form mother (NT) categories from the input consumed. The LR parser is driven over the input using shift and reduce actions, hence the parsers are a type of *shift-reduce automata*.

- *Stack:* differs in structure and use from the stack described in §2.2.2. Instead of using a list structure, the stack is implemented as a last-in-first-out (LIFO) queue. That is, symbol-state pairs are removed (POP operations) as well as added (PUSH operations) to the top of the stack. The POP operations occur when a reduce action is applied. We first remove the daughters of the rule before we add the newly created mother category of the rule to the stack.

- *LR table:* differs from the transition table as it also encodes reduce actions. The LR table consists of two parts; an *action* and *goto* table. The action table is similar to the transition

table, where rows represent states and columns represent terminals of the grammar. Cells contain competing parse actions given the current state S_c and terminal symbol. The goto table encodes the next state after a reduce action is applied.

- *Output:* we instead return the corresponding parse tree, whose root category spans the entire input (if found). Thus, we implicitly accept input if we can find such a derivation that is determined from the corresponding complete path through the underlying DFA.

2.3.2 Types of LR Parsers

There are several types of LR parse tables, where each type results in creation of the corresponding LR parser type, and components of the LR parser are otherwise the same as in Figure 2.4. Aho *et al.* (1986) describe three techniques to construct an LR(k) parsing table for a grammar; simple LR (SLR), lookahead LR (LALR) and canonical LR. These methods represent increasing left context used to make parsing decisions. Similarly, the variable k represents the number of *lookahead items*, the unexpended terminals in the input sequence, used as right context. As the level of context (the LR method and k) increases, so too do the number of states in the DFA. The LALR(1) method is popular as it provides a trade-off between accuracy (context) and efficiency (number of LR states). We describe LALR(1) parser construction and application herein, as we apply such an LALR(1) parser, modified to handle an ambiguous unification-based grammar.

2.3.3 Parser Actions

The types of actions applied by an LR parser include those considered previously for a recogniser. We define these three action types informally described thus far, as well as a fourth action type which applies a grammar rule.

LR Parser Configuration

We represent the *configuration* of the parser using a pair; the current stack state and unexpended input. For example consider the following configuration:

$$(S_0X_1S_1X_2S_2...X_mS_m, a_ia_{i+1}...a_n)$$

We denote categories of the grammar (terminal and nonterminal) using the variable X. This configuration corresponds to being in state S_m, i.e. $S_c = S_m$, the next input symbol is a_i and the input consumed thus far is $(a_1...a_{i-1})$.

The *action* Function

We refer to the unexpended input as the set of *lookaheads*, and the next input token as the *current lookahead* (or simply *lookahead* for short): la_c. In the recogniser described previously, we determine the next state using the function *action* which returns the next state given the current state and lookahead: $action(S_c, la_c)$. The next state specified results from one of three actions: (i) accept the input, (ii) reject the input or (iii) consume the input item a_i and move to the next state specified by *action*, i.e. $S_c = action(S_m, a_i)$. The final action type is referred to as a *shift* action in LR parsing. These three action types are encoded in the action table of the LR table.

Reduce Actions

In LR parsing, the fourth action type, a *reduce* action, requires different treatment of the configuration's stack. A reduce action corresponds to applying one of the rules (productions) in the grammar $A \rightarrow \beta$. Thus we form the category A by combining daughter categories β already created, where $\beta = \{D_1, D_2, ..., D_p\}$. The stack must contain these categories formed in order, that is, with D_p on top of the stack. We remove these daughter categories from the stack and place the newly created mother category A, and the next state, on the stack. Note that for a reduce action, we do not consume the next input symbol.

For example, if the grammar contains the rule `VP → V NP` and the stack contains a `NP` on top and then a `V`, we first remove these two items from the stack before placing the new `VP` analysis and next state on the stack. Thus, as the stack is implemented as a LIFO queue, a POP operation removes the daughters of a grammar rule, while PUSH adds the newly created mother category on the stack.

For a reduce action, the next state to visit depends on the state exposed after popping the daughter categories from the stack, the *ancestor* state, and the newly formed mother category A. This information is encoded in the goto table of the LR table, which we describe further in subsequent sections. For now, we consider the function $GOTO$ to return the next state to visit, given the ancestor state and mother category.

Configurations for each Action Type

The type of action returned by $action$, results in different parser configurations, as follows:

- *Shift actions*: if $action(S_m, a_i)$ = shift S_j, then the parser executes a shift action and moves to state j, consuming an input item as we did in the recogniser, entering the configuration:

$$(S_0 X_1 S_1 X_2 S_2 ... X_m S_m a_i S_j, a_{i+1} ... a_n)$$

- *Reduce actions:* if $action(S_m, a_i)$ = reduce $A \rightarrow \beta$ then we execute the reduce action by popping off the p daughters of the production from the stack, exposing the state S_{m-p}. Assuming $S_j = GOTO(S_{m-p}, A)$ then the resulting configuration will be:

$$(S_0 X_1 S_1 X_2 S_2 ... S_{m-p} A S_j, a_i ... a_n)$$

- *Accept:* if $action(S_m, a_i)$ = accept, then parsing is complete.
- *Reject:* if $action(S_m, a_i)$ = null, then the path terminates at S_m and we reject the input.

2.3.4 LR Table

Action and Goto Tables

The LR table consists of two parts, the action table and the goto table, and implicitly encodes the *grammar DFA*. The action table, encoded in a similar fashion to the transition table of a DFA, defines the set of possible actions given the current state S_c and the lookahead la_c. States correspond to rows and lookaheads to columns, while cells in the table contain parsing actions. In the goto table, the rows and columns correspond to the ancestor state and the nonterminal category created, respectively.

Grammar G_1

The unambiguous grammar G_1, shown in Figure 2.5, is a simple CFG which models prepositional phrases (PP), noun phrases (NP) and verb phrases (VP), i.e. NT={PP, NP, VP, S} over the terminals verb, pronoun and preposition: Σ={V,Pro,P}. The original root category of the grammar was S. However, to handle the case where more than one root category is specified, the *augmented grammar* instead defines a single root category T and new rules (for each of the previous root categories where each rule rewrites the new root category as the old category).

```
r0: T -> S
r1: S -> NP VP
r2: NP -> Pro
r3: PP -> P NP
r4: VP -> V NP
r5: VP -> V NP PP
```

Figure 2.5: Grammar G_1. Each rule number, using the format rx, refers to production number x.

The single top category of an augmented grammar (e.g. T) allows the resulting LR parser compiled over the grammar to indicate when the input should be accepted. That is, the input is acceptable if an action calls for the creation of this top category. Thus, we consider the LR parser to consist of a single accept state S_{acc}. Grammar augmentation is performed during construction of the LR table, that is, as part of the precompilation process.

Grammar G_1: LR Table

The corresponding LR table for G_1 is shown in Table 2.2, where si specifies a shift to state i while rj specifies a reduce by production number j of the grammar. The 'accept' action and empty cells specify that we should accept and reject the input, respectively. The goto table simply encodes the next state number given the ancestor state (row) and newly create mother category (column).

Grammar G_1: DFA

The corresponding *grammar DFA* for G_1 is shown as a transition graph in Figure 2.6. Each vertex corresponds to a state in the LR table and the vertex number illustrates the corresponding state number. Vertexes may also specify, in brackets, the ancestor state which must have been exposed to enable the transition to the state. Edges are compressed and labelled with the set of lookahead items for the edge (i.e. a single edge corresponds to each lookahead item) and also the action which is performed.

This transition graph provides a simplified view of the LR parser because the graph is conditionally traversed across a set of edges from a state that specifies the same reduce rule. That is, although multiple edges are shown for the same lookahead(s) and reduce rule, only one is applicable given the ancestor state exposed. Consequently, the automata is deterministic.

Relationship between the Grammar DFA and LR Table

The grammar DFA is constructed from the table in three stages as follows:

- *Vertex creation:* Each row in the table represents a state. As a result, we create one vertex in the graph for each state in the table. If the vertex's number is defined in a cell in the

state	action				goto			
	$	P	Pro	V	S	NP	VP	PP
0			s3		9	1		
1				s2			8	
2			s3			4		
3	r2	r2		r2				
4	r4	s5						7
5			s3			6		
6	r3							
7	r5							
8	r1							
9	accept							

Table 2.2: LALR(1) table for G_1.

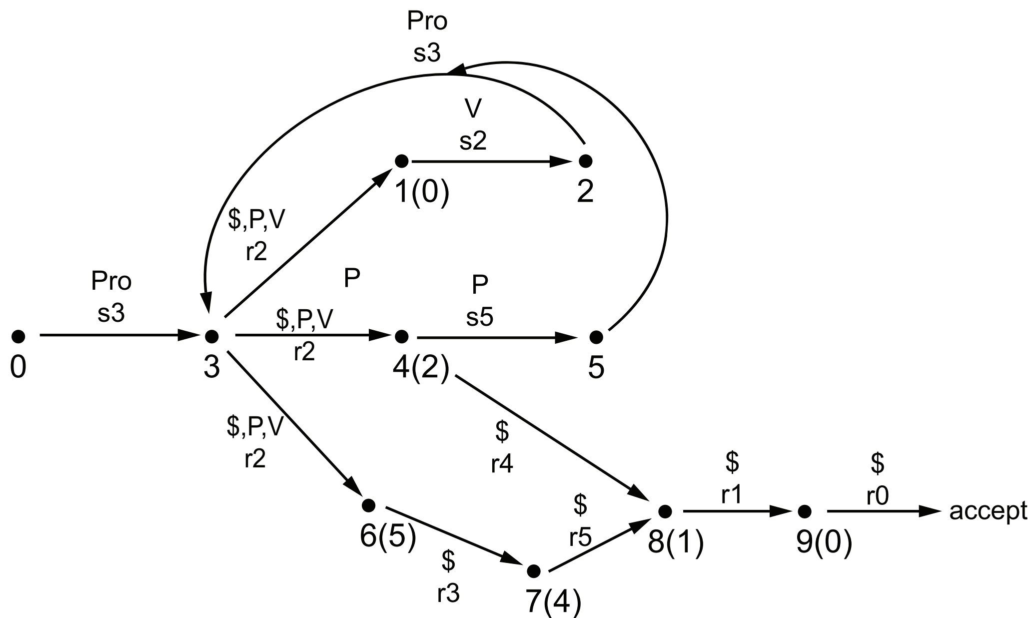

Figure 2.6: DFA for grammar G_1.

goto table for state (row) S_a, then we label the vertex with the ancestor state S_a. This illustrates that a reduce action should have exposed S_a on the stack to enable transition along an edge to this vertex in the DFA.

- *Shift edge creation:* Each shift in the action table sj for state (row) S_i in column Σ_k specifies that a directed edge should be created from state i to state j with label Σ_k.

- *Reduce edge creation:* For each reduce action in the action table rj, specifying a reduction of rule $A \to \beta$, for state (row) S_i in column Σ_k results in creation of one or more edges. We create a directed edge from state i to each state S_l, where S_l appears in the column for

category A in the goto table.

2.3.5 Parsing Program

The Parsing Process

Parsing determines paths through the underlying DFA, as described previously, from the start state S_0 to the accept state S_{acc} given the current input. To begin parsing, we first initialise the variable representing the current state to the start state. That is, we set $S_c = S_0$ and also set the current lookahead la_c to the first input token. The parsing program is implemented by extending the algorithm for simulating a DFA, as defined in §2.2.2. Lines 2–4 of the algorithm are replaced by statements conditioned on whether the action returned by $action(S_c, la_c)$ is a shift or reduce. The next input symbol is only consumed if the action is a shift. The stack is updated for each action type as described in §2.3.3.

Parsing over G_1

Table 2.3 illustrates the stack configurations reached after each action is applied for the G_1 LALR(1) parser over the sentence 'he likes her' with corresponding tagged input sequence: {`he_Pro likes_V her_Pro $`}. The resulting parse tree is extracted from the stack by keeping track of which rules applied in order and the word span of each category. In this example, the resulting parse tree is:
`(S (NP he_Pro) (VP likes_V (NP her_Pro)))`.

number	stack	input	action
1	0	Pro V Pro $	shift to state 3
2	0 Pro 3	V Pro $	reduce using r2
3	0 NP 1	V Pro $	shift to state 2
4	0 NP 1 V 2	Pro $	shift to state 3
5	0 NP 1 V 2 Pro 3	$	reduce using r2
6	0 NP 1 V 2 NP 4	$	reduce using r4
7	0 NP 1 VP 8	$	reduce using r1
8	0 S 9	$	accept

Table 2.3: Stack configuration for G_1 over PoS tag sequence {`Pro,V,Pro`}. The first column shows the configuration number while the next two columns form the tuple of each configuration reached. The final column illustrates the next action determined using the stack top state (S_c) and current lookahead (the first token in the input column).

2.3.6 Table Construction

LR(k) Parsing and Handles

The LR precompilation process results in compiling the states of the LR(k) table and the actions possible given the current state and next k lookaheads (including 0, for SLR parsers). This process facilitates a parsing mechanism capable of identifying the *handle*; the appropriate substring to reduce, along with the rule which should be applied to perform the reduction. Thus, LR tables encode additional context over the underlying CFG. Left-context is incorporated as states represent the handle, while right-context is incorporated in the set of k lookaheads.

For an LR(k) table, we identify which handles are appropriate *given* the next k input symbols as right context. In the shift-reduce LR parser, the state on top of the stack encodes all the information required to determine the current handle (or set of handles if the method is generalised to apply over an ambiguous grammar).

LALR(1) States

In an LALR(1) parser, each state represents the same (left) input having been consumed; either a terminal category of the grammar or a nonterminal of the category with the same left-context (category). For example, in G_1, `NP` may appear as the subject (in rule $r1$), the object of a verb (in rules $r4$ and $r5$) or within a preposition phrase (in rule $r3$). Consequently, there are three corresponding states for `NP` created in each of these contexts: states 1, 4 and 6, respectively. A trivial method to determine how many contexts each nonterminal category may appear in, involves counting the number of states that appear within the corresponding column for this category in the goto table.

LR(k) Items

Algorithms to construct the LR table from the grammar differ depending on the type of LR table, that is, the type of LR parser we wish to construct. In general, we create *LR(k) items*, tuples in the form $[A \rightarrow \alpha \cdot \beta, \delta]$, where the first element is a production of the grammar (with a dot at some position within the daughters of the rule) and δ is a set of k (including 0) terminal symbols of the grammar. The position of the dot indicates the input that has been witnessed to date; daughters to the left of the dot have been seen while daughters to the right of the dot may yet be witnessed (i.e. possible given the grammar).

LR(0) Items

To create the LR states of a grammar, we first create all LR(0) items. That is, we create an item for each rule in the grammar for each possible position of the dot (including start and end positions). The LR(0) items for the rule `S → NP VP` are shown in Figure 2.7.

```
S → · NP VP
S → NP · VP
S → NP VP ·
```

Figure 2.7: LR(0) items for the rule `S → NP VP`.

Kernel Items

We can define *kernel* (and conversely, *nonkernel*) items as those items with the dot in a position other than immediately to the right of the arrow (→). In addition to these items, we also consider the nonkernel items for the augmented top category (e.g. `T → · S`) as kernel items.

FIRST* and *FOLLOW

If the dot appears at the end of the production (e.g. $A \rightarrow \alpha\beta\cdot$) then this indicates that a reduce rule is possible at this point during parsing. For example, the third LR(0) item in Figure 2.7 indicates that an `NP` is followed by a `VP`, so we reduce to form the mother category `S`. Alternatively, if $A \rightarrow \alpha \cdot \beta$ is an item where $\beta \in \Sigma$ then we perform a shift over terminal β. We utilise the functions *FIRST* and *FOLLOW* to determine this kind of information from the item sets

where $FIRST(X)$ and $FOLLOW(X)$ indicate the terminal symbols that the symbol X may begin with or be followed by in the given grammar. For example, in G_1, $FIRST(\texttt{NP}) = \{\texttt{Pro}\}$ while $FOLLOW(\texttt{NP}) = \{\texttt{V},\texttt{P},\$\}$.

Collections of Item Sets: States

The precompilation process begins by augmenting the grammar and determining the set of all possible items given the CFG. Items are then grouped in sets which give rise to the states of the parser. During construction of these item sets, we effectively determine the *compiled DFA* for the grammar, as each new item set is created from the existing set by consuming the next terminal or nonterminal category to appear. From the resulting compiled DFA, we determine the set of LR states to be the collection of item sets. The actions of the LR parser are also encoded in this DFA.[2.3]

closure and *goto*

In order to group the item sets we require two functions: *closure* and *goto* (where this *goto* function is distinct from the uppercase *GOTO* defined previously). The function *closure* is defined over a set of items I, where we include each item of I in $closure(I)$, and expand each item in I to include all the nonkernel items for categories immediately to the right of the dot in the item. We repeat this expansion over the items until all such nonkernel items have been included. Formally, given $A, B, C \in NT$ and $\alpha, \beta, \gamma, \rho \subseteq \{\Sigma, NT, \varepsilon\}$, then if $A \rightarrow \alpha \cdot B\beta$ is in the closure of I then so too is $B \rightarrow \cdot C\gamma$ as is $C \rightarrow \cdot\rho$ and so on and so forth. As the set of nonkernel items for any given category is computed (or for efficiency, precomputed) for each category of the grammar, we compress the set of items in a closure so that each item set is represented only by the kernel items. However, in the following examples we represent each item set in full, showing both kernel and nonkernel items of the set.

The second function, *goto*, is defined in terms of the *closure* operation. Simply, $goto(I,X)$ represents the items which result if items in the set I receive a grammar symbol X as input. Thus, $goto(I,X)$ is defined to be the closure of the set of all items $A \rightarrow \alpha X \cdot \beta$ such that $A \rightarrow \alpha \cdot X\beta$ in the set I.

Constructing the Canonical LR(0) Item Sets

To construct the canonical collection of LR(0) items, we initialise the set of items Θ by taking the closure of the top category kernel items with the dot at the very start, $\texttt{T} \rightarrow \cdot\ \texttt{S}$. We repeatedly determine the $goto(I,X)$ for each $I \in \Theta$ and $X \in \{\Sigma, NT\}$ adding the resulting $goto(I,X)$ item to a set in Θ if they are already present in this set. If not, we create a new item set which includes the new item. For each new item set we also perform the closure operation over this set I to augment the item set with the nonkernel items. The set Θ forms our set of states in the compiled DFA, where each state produced by $goto(I,X)$ has an edge to it from the state corresponding to I labelled with the category X.

Canonical LR(0) Collection for G_1

The canonical LR(0) collection of sets for grammar G_1 is shown in Table 2.4. The table shows the set number j as I_j, and the corresponding kernel and nonkernel items of each set in the second and third columns, respectively. The remaining columns illustrate the next item set

[2.3] Note that the compiled DFA differs from the one we extract from the resulting LR table previously discussed (the grammar DFA). The algorithm for compiling the LR table creates the compiled DFA which contains edges *from* ancestor states rather than from the current state conditioned on the ancestor state.

created by determining the *goto* of the item (row) and grammar symbol (column). For example, *goto*(I_0:`NP →· Pro`,`Pro`)=I_3. That is, from an item in set I_0 we consume a *Pro* terminal symbol and move to an item in I_3. Starting from closure(`T →· S`), we create the first item set I_0. From here, we calculate the *goto* of I_0 with each X symbol in the grammar. For *goto*(`S →· NP VP`, `NP`) we create the item `S → NP · VP` which is currently not in any set in the collection. In view of this new item, we create a new set I_1. We then perform the closure operation over `S → NP · VP` in this new set to complete it. We continue until the *goto* function is unable to create a new set from the current sets.

	Items		goto(I,X)						
Set	Kernel	Nonkernel	S	VP	NP	PP	V	Pro	P
I_0	T →· S		I_9						
		S →· NP VP			I_1				
		NP →· Pro						I_3	
I_9	T → S ·								
I_1	S → NP · VP			I_8					
		VP →· V NP					I_2		
		VP →· V NP PP					I_2		
I_2	VP → V · NP				I_4				
	VP → V · NP PP				I_4				
		NP →· Pro						I_3	
I_3	NP → Pro ·								
I_4	VP → V NP ·								
	VP → V NP · PP					I_7			
		PP →· P NP							I_5
I_5	PP → P · NP				I_6				
		NP →· Pro						I_3	
I_6	PP → P NP ·								
I_7	VP → V NP PP ·								
I_8	S → VP NP ·								

Table 2.4: The canonical LR(0) sets for G_1.

Constructing a SLR Parsing Table

To create the SLR parse table for a grammar, we proceed in full as follows for augmented grammar G:

1. Construct the LR(0) canonical item sets $\Theta = \{I_0, I_1, ...I_n\}$ for G.

2. State i is constructed from I_i, and the table is initialised to contain empty cells for each terminal and nonterminal in the action and goto tables, respectively.

3. The corresponding parsing actions for the state are determined within the action table as follows:

(a) If I_i contains $A \rightarrow \alpha \cdot a\beta$, where a is a terminal, and $goto(I_i, a) = I_j$ then we set the cell in the action table for state i and terminal a to contain a shift to S_j, i.e. $action(S_i, a) = \mathrm{s}j$.

(b) If I_i contains rule number k with the dot at the end e.g. $A \rightarrow \alpha\cdot$, then set each cell in the action table for state i and for each terminal in $FOLLOW(A)$ to contain a reduce by rule k i.e. rk.

(c) If `T` $\rightarrow$ `S` $\cdot$ is in I_i this specifies to reduce to the augmented top category. As a result, we set the action in the cell for state i and terminal \$ to be 'accept'.

4. The goto table consists of transitions for state i and nonterminal A such that if $goto(I_i, A) = I_j$ then $GOTO(S_i, A) = S_j$.

5. We set the initial state of the parser to I_0, i.e. S_0, determined by the initial closure operation over the root category item e.g. `T` $\rightarrow \cdot$ `S`.

Constructing the Canonical LR(1) Parse Table

To create a canonical LR(1) table, we modify the *closure* and *goto* functions to operate over LR(1) items to create the item sets. The LR(1) items are designed to incorporate some right context to guide the parsing decisions. For the LR(1) item: $[A \rightarrow \alpha\cdot, a]$, we only reduce by $A \rightarrow \alpha$ if the next input symbol is a. The important distinction between this parser and the SLR parser described previously, is that the set of such a symbols will now be a subset of $FOLLOW(A)$ (recall that we allowed the reduction in the SLR table for all a in $FOLLOW(A)$).

States created over LR(1) items effectively split the states of the corresponding SLR states (constructed over LR(0) items) so that each state not only contains the information required to determine the current handle, but also indicate exactly which lookahead (input) symbols can follow the handle α, for which there is a possible reduction to the nonterminal A.

In G_1, a reduction to form `NP` by $r2$ occurs for the lookahead \$ only within the context of a `PP` or `VP` in rules $r3$ and $r4$, respectively. For the `NP` in subject position however, the lookaheads may be either of {`P, V`}. So two separate LR(1) item sets now correspond to the reduce to `NP` rather than the single LR(0) item `NP` $\rightarrow$ `Pro` $\cdot$. The LR(1) item sets corresponding to these situations are [`NP` $\rightarrow$ `Pro` $\cdot$, \$] and [`NP` $\rightarrow$ `Pro` $\cdot$, {`P, V`}], respectively.

Constructing the LALR(1) Parse Table

LALR(1) parse tables contain the same number of states as the SLR table, by compressing the LR(1) item sets i.e. those states found for the canonical LR(1) table. We can compress the LR(1) item sets into sets with the same *core*. Compressed sets contain sets of items in which only the lookahead values differ. These sets are used to create the states in the LALR(1) table. However, creation of the LR(1) item sets can prove infeasible for large grammars and a more efficient algorithm for creating the LALR(1) table involves creation of the LR(0) item sets (represented using kernel items only) which are then augmented with the set of lookahead symbols (the second element of the tuple).

In order to augment the LR(0) item sets with lookahead symbols we can generate lookaheads *spontaneously* or they may *propagate* from one LR(0) item set to the item set's *goto* item sets (described by Anderson *et al.* 1973, also Aho *et al.* 1986). The algorithm utilises a dummy lookahead symbol # to detect the situations in which lookaheads propagate, calling for multiple passes over the kernel items to perform the lookahead augmentation. If $[B \rightarrow \gamma \cdot C\delta, b]$ is in I, and we know for each nonterminal C the set of A, where each A is a left-most NT for all possible

right-most rule expansions ($\Rightarrow^*_{rm}$) of C i.e. where $C \Rightarrow^*_{rm} A\eta$, then we can determine the values of a for which $[A \rightarrow X \cdot \beta, a]$ is in $goto(I,X)$ as the set $FIRST(\eta\delta)$. Note that this will be a subset of $FOLLOW(A)$, the set given A is derived (expanded) from C (and not another category of the grammar). If the set $\eta\delta$ is null or may be expanded to ε then b is also a possible lookahead and $[A \rightarrow X \cdot \beta, b]$ will be in the $goto(I,X)$. In this case, we consider that the lookaheads (i.e. b) propagate from $B \rightarrow \gamma \cdot C\delta$ to $A \rightarrow X \cdot \beta$.

Constructing the G_1 LALR(1) Parse Table

G_1 is a simple grammar. As a result, the SLR and LALR(1) parse tables (Table 2.2) are equivalent. This occurs as all nonterminal categories are each expanded from a single nonterminal category. That is, continuing the discussion in the previous section, the following holds for all A: $FIRST(\eta\delta) = FOLLOW(A)$. Thus we derive the LALR(1) table for G_1 either by applying the SLR table construction method from the item sets (states) in Table 2.4, or by augmenting these item sets with lookaheads.

2.4 GLR Parsing

We now describe modifications to the LR parser to facilitate nondeterminism in the parsing actions. In particular, we describe the modifications made by Tomita (1987) that generalise LR parsing, to compile and apply an LR parser over an ambiguous CFG. Tomita describes *generalised LR parsing* (GLR, henceforth), an extension of the LR parsing framework, to allow parallel tracking of multiple state transitions and stack actions by using a *graph-structured* stack. Thus we allow for multiple paths through the NFA to be determined, and each complete path found corresponds to a derivation as discussed in §2.2.1. We describe these modifications over an LALR(1) parser.

2.4.1 Relationship to the LR Parsing Framework

If we consider the LR framework in Figure 2.4, the modifications that Tomita (1987) describes may be summarised as follows:

- *LR table:* can specify more than one possible action i.e. sets of transitions as in the NFA described previously. Thus cells in the LR table now specify a set of possible actions and *action* returns this set given the current state and lookahead.
- *Stack:* we generalise the standard linear LIFO queue structure to enable the stack to merge and diverge as the corresponding paths (state sequences) do so.
- *LR parsing program:* processes each path in parallel, so that all possible paths are found throughout the underlying NFA.
- *Output:* the set of all possible parses licensed by the grammar. That is, the parser outputs one parse for each complete path found.

2.4.2 Table Construction

The LR table is constructed as described previously, though more than one action can be applied given the current state and lookahead. Consequently, the function *action* returns any number of possible shift and reduce actions. This results in action *conflicts* or *competing* actions in cells throughout the LR table. Shift-reduce or reduce-reduce conflicts are possible, though shift-shift conflicts are not, as la_c has a single value which determines a single possible shift move.

Competing actions correspond to either ambiguities in the grammar (that each result in competing derivations) or to alternative derivations that are not distinguishable given the current context (so that one or more actions result in a path that eventually terminates).

Grammar G_2 Example

Grammar G_2, shown in Figure 2.8, extends Grammar G_1 and models simple attachment preferences for prepositional phrases (`PP`) between noun phrases (`NP`) and verb phrases (`VP`). The top category `T` has also been added, so that the augmented grammar also contains `T'` and the rule $r0$: `T'` $\rightarrow$ `T`. Rule $r0$ specifies the edge to the accept state.

The GLR table (generalised LALR(1)) for this grammar is shown in Table 2.5. The corresponding grammar NFA is shown in Figure 2.9. There are several cells in the table which define multiple actions. For example, for state 5 with a preposition lookahead we have two possible actions: $action(5, \texttt{P}) = \{r5, s4\}$. This represents the situation where an `NP` occurs after a terminal `P` (as the ancestor state is 4) and the lookahead is `P` as well. So we have the choice between creating a `PP` using the previously seen `P` and `NP` with $r5$: `PP` $\rightarrow$ `P NP`, or shifting over the next `P`. In the latter case, we may include the current `NP` with the next preposition phrase, using either $r4$ or $r7$.

```
r0: T' -> T
r1: T -> S
r2: S -> NP VP
r3: NP -> Pro
r4: NP -> NP PP
r5: PP -> P NP
r6: VP -> V NP
r7: VP -> V NP PP
r8: VP -> VP PP
```

Figure 2.8: Grammar G_2.

2.4.3 Graph-structured Stack

If a single derivation is possible, then a single path can represent the derivation. Otherwise, the stack diverges to separate paths where more than one action is applicable and merges when the same state is reached for the current lookahead (as from this point the same actions will be processed). This equivalence is critical as it allows for subsequent parsing actions to be applied to a single entity. Consequently, it allows for tractable computation over large ambiguous grammars.

The graph-structured stack (stack, henceforth) forms an NFA that is a subset of the grammar's NFA implicit in the GLR table. Each vertex in the stack now holds more information in addition to the state number of the vertex. The additional information includes the resulting syntactic structure(s) of the sentence, enabling the parsing program to return all full derivations possible given the grammar. That is, the parsing algorithm is an *all-path* parsing algorithm in that it determines all possible paths in the underlying NFA, that is, all possible derivations.

state	action				goto				
	$	P	Pro	V	T	S	NP	VP	PP
0			s2		12	1	3		
1	r1								
2	r3	r3		r3					
3		s4		s7				10	6
4			s2				5		
5	r5	r5 s4		r5					6
6	r4	r4		r4					
7			s2				8		
8	r6	r6 s4							9
9	r4 r7	r4 r7							
10	r2	s4							11
11	r8	r8							
12	accept								

Table 2.5: Generalised LALR(1) table for G_2.

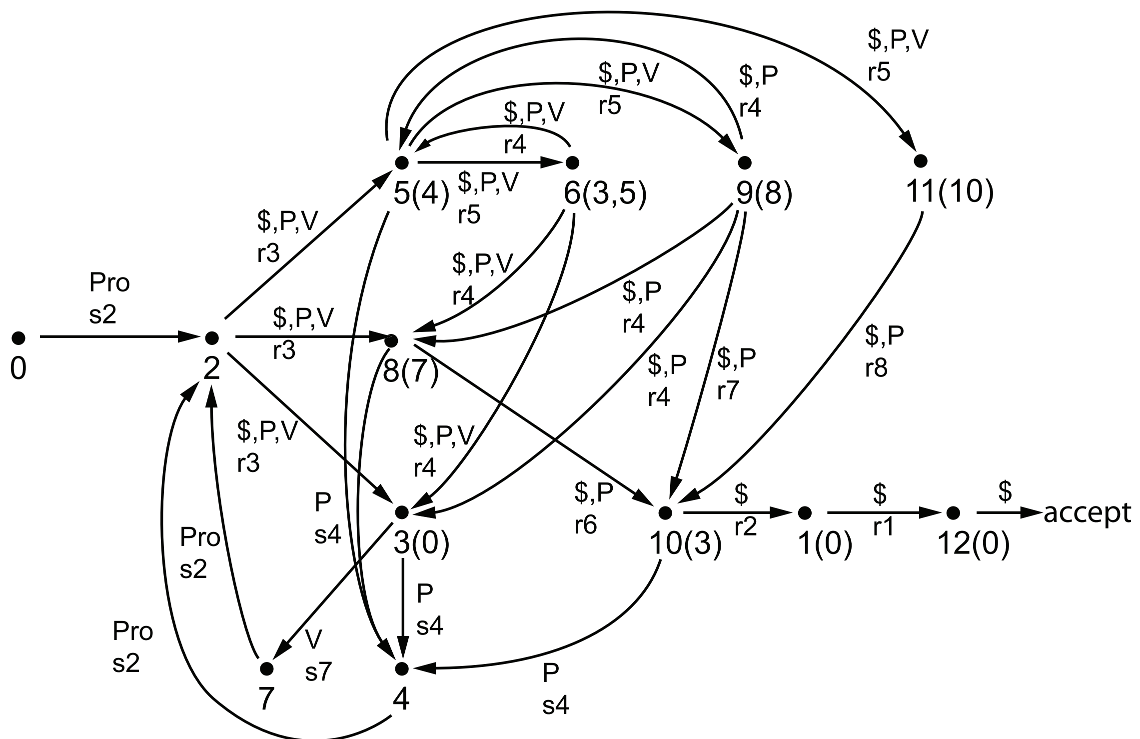

Figure 2.9: Grammar NFA for G_2.

Example Stack over G_2

Given the sentence 'He gives it to her' with corresponding PoS tag sequence {Pro,V,Pro,P,Pro} the possible parses, and corresponding state sequences through the NFA, are shown in Figure 2.10. The stack over this example is shown in Table 2.6 in a similar fashion to the stack shown for the unambiguous grammar G_1 in Table 2.3. However, the *stack* column is now subdivided into four columns. The first illustrates the stack shared by all three parses. The second column illustrates the stacks that each continue from the stack in the first column, while the third illustrates the stack that is again shared between all parses. That is, moving from the first to second column, the stack diverges (splits). Moving from the second column to the third, shows these stacks merge again. The parse number is shown in the fourth stack column, which illustrates the parses that result for the given stack.

```
P1:(T (S (NP Pro)
         (VP V (NP Pro)
               (PP P (NP Pro)))))
0,2,3,7,2,8,4,2,5,9,10,1,12,acc
P2:(T (S (NP Pro)
         (VP (VP V (NP Pro))
             (PP P (NP Pro)))))
0,2,3,7,2,8,10,4,2,5,11,10,1,12,acc
P3:(T (S (NP Pro)
         (VP V (NP (NP Pro)
                   (PP P (NP Pro))))))
0,2,3,7,2,8,4,2,5,9,8,10,1,12,acc
```

Figure 2.10: Example parses for G_2, with corresponding state sequences that produced the parse. The state *acc* represents the accept state.

Figure 2.11 illustrates the same stack diagrammatically as a NFA. In this figure, the stack is separated into word boundary stacks that illustrate the parsing actions (edges) performed for each lookahead. Edges are arrows shown with solid lines. Each edge corresponds to a parsing action, which results in moving from one state to the next. For a reduce action the lookahead is not updated. Thus, reduce actions result in subanalyses in the same word boundary stack and correspond to downward edges in this figure. In contrast, for a shift action we consume the current lookahead, and a horizontal edge illustrates the transition to the next word boundary stack. We perform all possible reduce actions prior to performing the possible shift actions for a given lookahead. This order is evident in the edge numbering.

For reduce actions (downward edges), we create a mother category from daughter subanalyses represented within vertexes of this graph. A dotted line between vertex i and vertex j illustrates that the ancestor state i is exposed when we perform a reduce action (POP the daughter categories from the stack) that results in a subanalysis for vertex j. The newly created mother category (shown in brackets) in vertex j spans the subset of the input shown from vertex i to vertex j. For example, given lookahead V in word boundary stack 1, we reduce from state 2 to 3 with $r3$ resulting in a NP. The word span of vertex 3 is from 0 to 1 (i.e. over Pro) and the ancestor state is in vertex 0 which represents the start state S_0.

The diverging paths in this graph, and in the stack example, illustrate the two positions

where the individual parses result. That is, the two nondeterministic decision points. The first decision point occurs at state 8 with lookahead `P` (stack configuration 6 in Table 2.6); where reducing to state 10 results in P2. Similarly, at state 9 with lookahead `$` (stack configuration 11b in Table 2.6) we decide between P1 and P3.

2.4.4 Parse Forest

As we build the stack during parsing, the compact representation of all possible parses, the *parse forest*, is also built within this structure. The parse forest is efficiently represented using *subtree sharing* and *local ambiguity packing* which we now describe.

Relationship between the Parse Forest and Graph-structured Stack

First, recall that the stack is separated into word boundary stacks, that is, one word boundary stack for each item (lookahead) in the input sequence. Within each vertex of a word boundary stack we encode the corresponding subanalyses (subtrees) for the vertex's LR state. That is, each subanalysis results from a parsing action that specifies the vertex's LR state as the next state (for the given lookahead).

Each subanalysis corresponds to a *node* in the parse forest. Thus each vertex in the stack represents a set of nodes in the parse forest for the given lookahead and LR state. Therefore, the parse forest and graph-structured stack are separate data structures, though the stack encodes the parse forest which represents the analyses built within the stack.

Subtree Sharing

If a vertex is formed after application of a shift action, then the corresponding nodes (subanalyses) for this vertex are *word nodes*. In contrast, after a reduce action, we form a subanalysis using one or more daughter nodes (in stack vertexes) and we create a *tree node*. A tree node represents a subanalysis (subtree), created by storing the newly created mother category as well as *pointers* to the daughter nodes (rather than copying each daughter node's subtrees). This process is known as *subtree sharing*, as different nodes may specify (that is, contain pointers to) the same daughter nodes in the parse forest.

Local Ambiguity Packing

If we create nodes (subanalyses) for a vertex, then a subset of these nodes may represent equivalent subanalyses over the same subset of the input sequence (that is, with the same word span). The definition of node equivalence varies between parsing systems and grammars. For a CFG, two subanalyses are equivalent if their mother categories are the same.

Equivalent nodes in a vertex represent competing nodes (subtrees) in the parse forest. As we create these nodes within the same state of the LR parser (with the same lookahead symbol and word span) we apply the same set of subsequent parsing actions to each. To reduce redundant processing, we need only apply these parsing actions once. In view of this redundancy we merge competing nodes in the parse forest (and therefore, in the vertex of the stack) through *local ambiguity packing* (packing, henceforth). Here, we represent each set of nodes using one node of the set. Subsequent parsing actions are then applied to the single representative node only.

2.4.5 LR Parsing Program

The LR parsing program proceeds in the same fashion as previously described. Although it now allows for more than one action to be applied given the current state and lookahead item.

number	from	stack				input	action
1		0			P1,P2,P3	Pro V Pro P Pro $	s2
2	1	0 Pro 2			P1,P2,P3	V Pro P Pro $	r3
3	2	0 NP 3			P1,P2,P3	V Pro P Pro $	s7
4	3	0 NP 3 V 7			P1,P2,P3	Pro P Pro $	s2
5	4	0 NP 3 V 7 Pro 2			P1,P2,P3	P Pro $	r3
6	5	0 NP 3 V 7 NP 8			P1,P2,P3	P Pro $	r6, s4
7a	6 (r6)	0 NP 3	VP 10		P2	P Pro $	s4
8a	7a	0 NP 3	VP 10	P 4	P2	Pro $	s2
8b	6 (s4)		V 7 NP 8		P1,P3		
9a	8a	0 NP 3	VP 10	P 4 Pro 2	P2	$	r3
9b	8b		V 7 NP 8		P1,P3		
10a	9a	0 NP 3	VP 10	P 4 NP 5	P2	$	r5
10b	9b		V 7 NP 8		P1,P3		
11a	10a	0 NP 3	VP 10 PP 11		P2	$	r8
11b	10b		V 7 NP 8 PP 9		P1,P3		r4, r7
12a	11b (r4)	0 NP 3	V 7 NP 8		P3	$	r8
13	11a,11b (r7),12a	0 NP 3 VP 10			P1,P2,P3	$	r2
14	13	0 S 1			P1,P2,P3	$	r1
15	14	0 T 12			P1,P2,P3	$	accept

Table 2.6: Stack configurations for G_2 over PoS tag sequence {Pro,V,Pro,P,Pro}. The first column shows the configuration number while the second column shows the previous stack configuration and which action was applied (in brackets) if more than one action was possible. The next two columns ('stack' and 'input') form the tuple of each configuration reached. The final column illustrates the set of possible actions, determined using the stack top state (S_c) and current lookahead (the first token in the input of column 4).

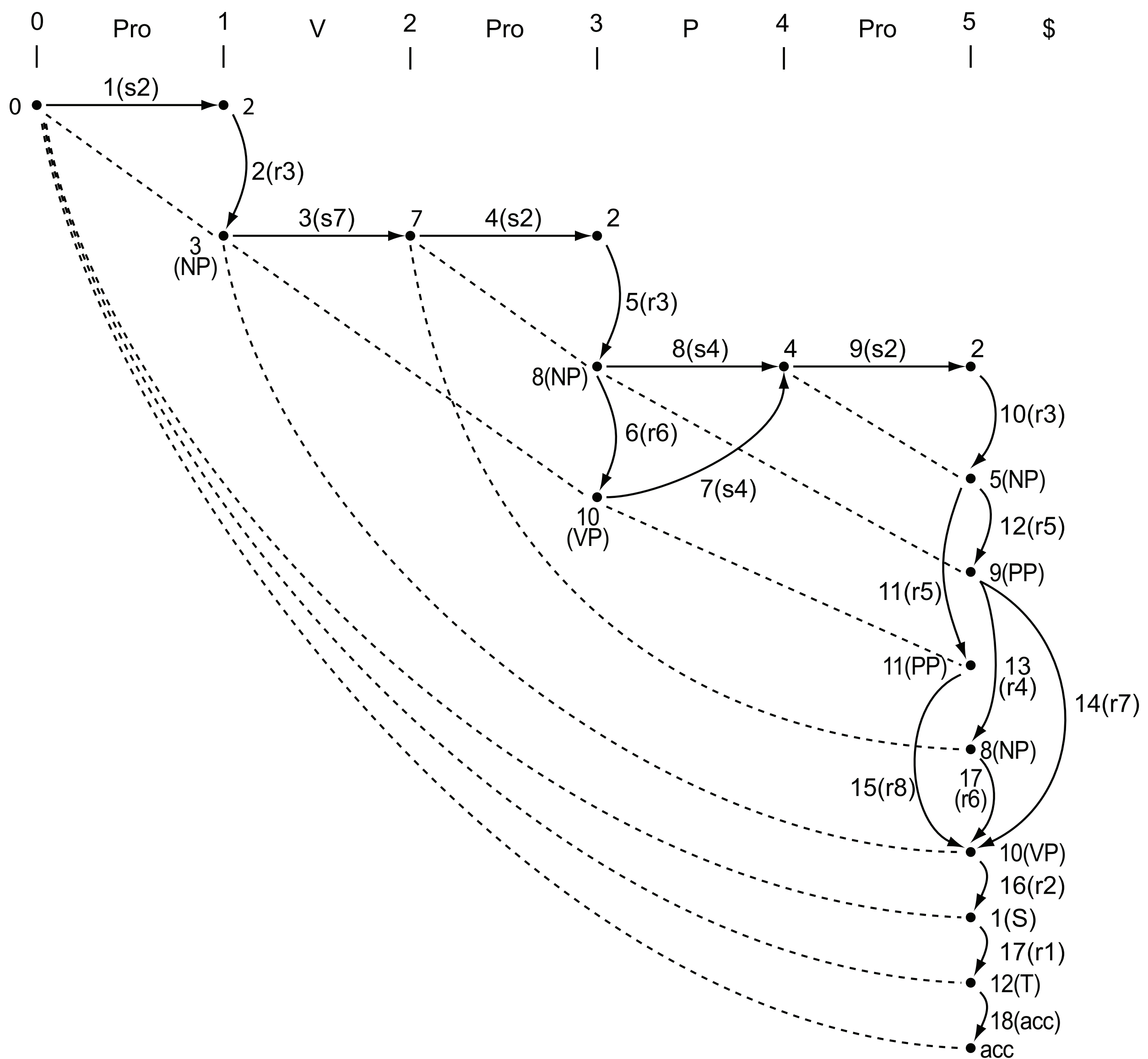

Figure 2.11: Example graph-structured stack for G_2 over PoS tag sequence {Pro,V,Pro,P,Pro}. Each vertex is shown with the corresponding state number and root category for each subanalysis (in brackets). For word vertexes, which are created by shift actions, the category is the word itself. The vertexes are separated into word boundary stacks (over each lookahead), where each word boundary stack number is shown across the top of this figure. To the right of each word boundary stack is an item in the input sequence, where this item is the lookahead for the boundary. We also number edges (solid lines) which show the order in which the actions are performed and, in brackets, the action to perform. Dotted lines illustrate the ancestor state exposed for a reduce action that creates a subanalysis for the vertex.

The parsing program begins by initialising the first word boundary stack (the 0th stack) to a list containing a single vertex data structure representing the start LR state. All other word boundary stack positions are initialised to empty. The program continues by applying all possible reduce actions for each vertex in the current word boundary stack, so that subsequent shift actions may be applied to all possible states in this stack (for the given lookahead).

Each production may be augmented with a set of tests, where each nonterminal is augmented with a set of attributes and the tests define whether the attribute values are acceptable. Tomita (1987) describes that during parsing, whenever a reduce action forms a higher-level nonterminal using a phrase structure rule, a separate function associated with the rule is called that defines, passes and/or tests the attribute values of the resulting nonterminal. If this function returns notification that one or more tests have failed, then the parser discards the resulting nonterminal.

The parsing program also includes a processing stage to enable extraction of the set of (n-best) derivations from the resulting parse forest. This code is considered to *unpack* the parse forest, given that the nodes (competing subanalyses) are *packed* in the data structure.

2.4.6 Output

Given subtree sharing and packing, the parse forest itself represents another NFA, where we may move from root node (representing the root category of the grammar) to word nodes. The path diverges where packing occurs and merges for subtree sharing. To unpack the parse forest and extract the set of competing derivations, we simply need to traverse the NFA and consider each possible combination of subanalyses for each node in the parse forest. That is, we perform a depth-first search of this structure and for each node we return the set of possible subanalyses. Optionally, this set is pruned so that only the n-best set of subanalyses is returned.

Given the set of subanalyses for each daughter node, we create an alternative subanalysis for the current node for each possible combination of daughter subanalyses with the given mother category. We combine this set of subanalyses with the set returned by any packed nodes, as this set represents the alternative subanalyses for the current node. At the root node of the parse forest, the set of possible analyses represent competing derivations for the given input sequence (sentence).

2.4.7 Modifications to the Algorithm

Kipps (1989) reformulates the modifications defined by Tomita (1987) to improve the parser's efficiency through changes to the state-popping process. The algorithm identifies, for a given reduce rule, each possible *ancestor*. That is, the set of daughters on the stack and the resulting ancestor vertex. The `ANCESTORS` function, which dominates the complexity of the parser, determines a set of possible ancestors given the current vertex, the number of daughters and the mother category we wish to create. This function is modified so that the reduce function utilises a lookup table, reducing the complexity of the parser from n^{p+1}, where p is the maximum number of daughters for a production, to n^3.

2.5 Statistical GLR (SGLR) Parsing

This section discusses existing approaches to incorporate statistical models over the GLR (LR, henceforth) parsing framework defined above. In particular, we define the different normalisation methods over the LR table, given action counts derived over a supervised training corpus.

2.5.1 Probabilistic Approaches

PCFG Model

At first, methods considered LR parsing as a purely operational mechanism, aiming to distribute probabilities originally associated with the probabilistic CFG (PCFG) (Wright & Wrigley, 1989). However, these methods fail to take advantage of the additional context available in the LR parser. The resulting level of context is equivalent to that available in the PCFG, which are acknowledged in the literature as inadequate due to a lack of context-sensitivity.

Stack Configurations

Su *et al.* (1991) propose a model that is moderately more context-sensitive than the underlying CFG. They distribute probability mass across possible action sets for the stack configurations that result for each lookahead item, i.e. between the possible word boundary stacks. The probability of a parse is the product of the probabilities for each stack configuration that results (prior to each shift action) during construction of the parse. Effectively, they include full context (stack configuration) in the probability model. However, this model requires a complex algorithm to train and is it is not efficient to decode.

Parser Actions

Briscoe & Carroll (1993) (B&C, hereafter) propose a method that distributes probability mass directly between competing parsing actions in the LR table, so that the parsing model successfully incorporates a greater level of context compared to the underlying CFG. Further, they estimate the probability distributions using simple MLE and the model is efficient to decode. The probability of each parse is the product of all shift/reduce actions that result in the parse. Inui *et al.* (1997) (I&T, hereafter) refine the B&C probabilistic model by providing an alternative normalisation method. Associating probabilities with each vertex in the stack is problematic, as vertex represent a set of subanalyses (nodes) where each results from a different reduce action. Instead, Carroll (1993) associates probabilities with each subanalysis and with each packed subanalysis, i.e. with each node in the parse forest, rather than with each vertex in the graph-structured stack (see §2.4.4).

2.5.2 Estimating Action Probabilities

Estimating action probabilities in the LR table consists of a) recording an action history for the correct derivation (for each sentence in a treebank), b) computing the frequency of each action over all action histories and c) normalising these frequencies to determine probability distributions over conflicting (i.e. shift/reduce or reduce/reduce) actions. Models differ in the last step, the normalisation method during MLE.

Action Counts

In order to estimate action probabilities, we must first derive from a supervised corpus the count for each action in the LR table. Given a pair (s,A), from an annotated treebank as defined in §1.3.1, we determine the parse forest for the sentence s. We then identify the parse in the parse forest that matches the tree A specified in the treebank. This 'correct' parse has a corresponding set of shift and reduce actions that results in creation of the parse. For each shift and reduce action in this set, we add a count of 1 to the count for this action in the LR table. Repeating this process for each sentence (training instance) in the treebank, we determine the total action count for each action in the LR table. We now discuss alternative normalisation methods to

form probability distributions within the LR table, and consequently, for parses in the parse forest.

Lookahead Normalisation

It seems reasonable to determine the probability of each parsing action through MLE over competing action sets i.e. within cells of the LR table. We refer to this normalisation method as *la-norm*. The resulting probabilities are analogous to the transition probabilities in first-order HMM PoS-taggers as we assign probability mass between transitions in the underlying NFA with the same edge label (lookahead item).

State and B&C Normalisation

B&C, by contrast, use MLE to estimate their model's parameters from counts over rows of the LR table, that is across all lookaheads for each state, which we refer to as *state-norm*. Their full normalisation method further distributes probability mass between the alternative states possible after a reduce action. Thus, they effectively distribute probability mass between *all* actions (edges) possible from a state in the grammar NFA, including the edges that are conditional on the ancestor state exposed after a reduce action. However, this distributes probability mass between actions that do not compete with each other given the current input, that is, lookahead item. As a result, probabilities for actions are deficient, i.e. the probability of all parses licensed by the grammar do not sum to 1.

Inui Normalisation

I&T propose an alternative normalisation method, based on whether the current state was reached from a shift or reduce action. In which case, they distribute probability mass between cells (la-norm) or across rows (state-norm), respectively. They motivate this method using the conditional probability of moving from the current stack configuration σ_{i-1}, to another, σ_i. They estimate this probability using the stack-top state S_c, next input symbol la_c and next action a_i. They argue that the conditional probability of the next lookahead is known after a reduce action (i.e. it is unchanged) and unknown after a shift. Let us consider S_s and S_r mutually exclusive sets of states, which represent those states reached after shift or reduce actions, respectively. The probability of a given transition from one stack state to another (that is, of the given action a_i) can be estimated using:

$$P(la_c, a_i, \sigma_i | \sigma_{i-1}) \approx \left\{ \begin{array}{ll} P(la_c, a_i | S_c) & S_c \in S_s \\ P(a_i | S_c, l_i) & S_c \in S_r \end{array} \right\}$$

Therefore, they normalise over all lookaheads for a state or over each lookahead for the state, depending on whether the state is a member of S_s or S_r, respectively.

I&T also remove subdivision of probability mass between goto ancestor states for a given nonterminal category. They argue that this decision is deterministic in their model, which is based on stack state rather than the top-state S_c. However, in practice they utilise S_c to represent the stack i.e. back-off to a purely state-based level of context. I&T describe refinements that are expected to provide performance gains over the model of B&C, because la-norm models the preferences between competing actions only. Moreover, state-norm implicitly incorporates bigram statistics in the model.

Example

Returning to our previous example over the grammar G_2, we consider a training corpus that consists of seven sentences: one example of P1, two of P2 and four P3 parses. Table 2.7 shows the total action counts in brackets next to each action in the LR table. We illustrate only the action part of the table as the goto table is unchanged from Table 2.5. A tuple in the second row illustrates the probabilities for state-norm, la-norm and the normalisation model by I&T, respectively. B&C counts and resulting probabilities (for each ancestor vertex) appear for reduce actions in the remaining rows for each state, with the ancestor state in brackets. We do not apply smoothing in this example. Although if no counts are seen for any of the competing actions in a set, we show the probabilities distributed equally between the unseen actions.

The probability of P1, P2 and P3 for each normalisation model are shown in Table 2.8. Given the training data, the resulting probability model should rank P1 $<$ P2 $<$ P3. However, while both lookahead and I&T based normalisations achieve this ranking, with probabilities in ratio to the number of each parse seen during training, both state and B&C normalisation methods do not. A similar example was shown by I&T, though over a different grammar. B&C and state based normalisation both rank P2 $<$ P1, which on closer inspection of both examples is primarily due to the subdivision of probability mass to noncompeting actions (at this point in parsing). That is, to those that are not one of the two decision points (from states 8 and 9) as discussed in §2.4.3.

2.6 RASP

RASP (the 'robust accurate statistical parser') is a robust statistical analysis system for English developed by Briscoe & Carroll (2002). RASP is a SGLR parser, applying the probability distribution over complete derivations, using the I&T probability model over parsing actions as described in §2.5.2. This section provides specific details of RASP's grammar in §2.6.1, training in §2.6.2, parser application in §2.6.3 and finally, we describe the output formats available in §2.6.4.

2.6.1 Grammar

Unification-based Metagrammar

Briscoe (2006) describes the manually written feature-based unification grammar, where terminals are defined over PoS tags (Elworthy, 1994, the CLAWS II tagset). The grammar is written in the ANLT formalism, i.e. as a *metagrammar*, based on the notion of generalised phrase structure grammar (GPSG). Terminals do not include the null category and features hold atomic values so that rules are written using only a subset of the attribute-valued (AV) grammar possibilities defined by the ANLT formalism (Grover *et al.*, 1993). The grammars developed for use in RASP are referred to as the set of 'tag sequence grammars', *tsg*, and version numbers are appended to this acronym. The grammars utilised in this work are variants of the final *tsg15* released with version 2 of RASP (Briscoe & Carroll, 2006).

Attributes can be organised into sets for the purpose of feature propagation (*feature sets*) or simply to enable abbreviated representation of common sets (*aliases*). For example, Figure 2.12 shows a grammar rule analysing a verb phrase followed by a prepositional phrase modifier. The rule's name is `V1/vp_pp` and the syntactic specification which follows on the first line contains alias categories `V1`, `H1` and `P2` which are defined in the grammar as:

```
ALIAS V1 = [V +, N -, BAR 1].
```

state	action			
	$	P	Pro	V
0 S_r			s2 (7) {1,1,1}	
1 S_r	r1 (7) {1,1,1} $7^{(12)}$ $1^{(12)}$			
2 S_s	r3 (7) {0.33,1,0.33} $0^{(3)},7^{(5)},0^{(8)}$ $0^{(3)},0.33^{(5)},0^{(8)}$	r3 (7) {0.33,1,0.33} $0^{(3)},0^{(5)},7^{(8)}$ $0^{(3)},0^{(5)},0.33^{(8)}$		r3 (7) {0.33,1,0.33} $7^{(3)},0^{(5)},0^{(8)}$ $0.33^{(3)},0^{(5)},0^{(8)}$
3 S_r		s4 (0) {0,1,1}		s7 (7) {1,1,1}
4 S_s			s2 (7) {1,1,1}	
5 S_r	r5 (7) {1,1,1} $0^{(6)},5^{(9)},2^{(11)}$ $0^{(6)},0.71^{(9)},0.29^{(11)}$	r5 (0) {0,0.5,0.5} $0^{(6)},0^{(9)},0^{(11)}$ $0^{(6)},0^{(9)},0^{(11)}$ s4 (0) {0,0.5,0.5}		r5 (0) {0,1,1}
6 S_r	r4 (0) {0.33,1,1} $0^{(3)},0^{(5)},0^{(8)}$ $0.11^{(3)},0.11^{(5)},0.11^{(8)}$	r4 (0) {0.33,1,1} $0^{(3)},0^{(5)},0^{(8)}$ $0.11^{(3)},0.11^{(5)},0.11^{(8)}$		r4 (0) {0.33,1,1} $0^{(3)},0^{(5)},0^{(8)}$ $0.11^{(3)},0.11^{(5)},0.11^{(8)}$
7 S_s			s2 (7) {1,1,1}	
8 S_r	r6 (4) (10,4) {0.36,1,1} $4^{(10)}$ $0.36^{(10)}$	r6 (2) (10,2) {0.18,0.29,0.29} $2^{(10)}$ $0.18^{(10)}$ s4 (5) {0.45,0.71,0.71}		
9 S_r	r4 (4) {0.8,0.8,0.8} $0^{(3)},0^{(5)},4^{(8)}$ $0^{(3)},0^{(5)},0.8^{(8)}$ r7 (1) {0.2,0.2,0.2} $1^{(10)}$ $0.2^{(10)}$	r4 (0) {0,0.5,0.5} $0^{(3)},0^{(5)},0^{(8)}$ $0^{(3)},0^{(5)},0^{(8)}$ r7 (0) {0,0.5,0.5} $0^{(10)}$ $0^{(10)}$		
10 S_r	r2 (7) {0.78,1,1} $7^{(1)}$ $0.78^{(1)}$	s4 (2) {0.22,1,1}		
11 S_r	r8 (2) {1,1,1} $2^{(10)}$ $1^{(10)}$	r8 (0) {0,1,1} $0^{(10)}$ $0^{(10)}$		
12 S_r	accept			

Table 2.7: Statistical models over the action table for G_2.

Parse	Normalisation method			
	State-norm	la-norm	B&C	I&T
P1	0.0025	0.1420	0.0018	0.0051
P2	0.0011	0.2900	0.0003	0.0104
P3	0.0036	0.5680	0.0026	0.0204

Table 2.8: Example of the parse probabilities that result for different LR parser normalisation methods.

```
V1/vp_pp : V1[MOD +] --> H1 P2[ADJ -, WH -] :
    1 :
    2 = [PSUBCAT NP], (ncmod _ 1 2) :
    2 = [PSUBCAT NONE], (ncmod prt 1 2) :
    2 = [PSUBCAT (VP, VPINF, VPING, VPPRT, AP)], (xmod _ 1 2) :
    2 = [PSUBCAT (SFIN, SINF, SING)], (cmod _ 1 2) :
    2 = [PSUBCAT PP], (pmod 1 2).
```

Figure 2.12: Example metagrammar rule. This rule definition shows the rule name and syntactic specification (on the first line), with semantic rules for the GR output format shown in the remaining lines.

```
ALIAS H1 = [H +, BAR 1].
ALIAS P2 = [V -, N -, BAR 2].
```

The tsg grammars utilise x-bar theory, expressed in the feature `BAR`. The `V` and `N` features represent the major categories of verb and noun, while the values `+` and `-` represent the presence and absence of the feature, respectively. The head daughter is identified using the feature `H`, so that in this example, the first daughter is the head daughter. In addition to these features, a number of feature-value pairs are defined in the syntactic specification. For example, the prepositional phrase daughter is a non-wh phrase.

Object Grammar

The metagrammar enables the grammar writer to express the rules in a compressed and manageable format e.g. with aliases. These ‘compressed’ rules are compiled into an *object grammar* by ‘expanding out’ the phrase structure rules with additional features. The resulting object grammar contains categories represented as lists, specifying the category number followed by values corresponding to the features for the given category. Each value is set to either a specific predetermined value (e.g. `+`), or a variable value (starting with `@`) which is instantiated during unification. A table holds the corresponding feature names for each category.

For example, the 6th category type in tsg15 corresponds to a verb category. The following example shows a verb with several features instantiated, with the associated entry in the feature name table as follows:

```
(6 - + |1| @2787 @2815 - PAST + - - -)
(N V BAR PLU MOD AUX VFORM FIN CONJ SCOLON QUOTE)
```

Rules in the object grammar consist of lists of such categories. The first item is the rule’s mother, followed by the rule name, while remaining categories represent the daughters of the

rule. Each feature value is instantiated as specified in the metagrammar rule and features that are required to unify between daughters, or pass from the head daughter to the mother, are specified by using the same variable number (e.g. @7) in the daughter's and mother's category lists. The `V1/vp_pp` rule shown in Figure 2.12 results in the compiled object grammar rule, with the features set as specified in the aliases and within the rule itself, as follows:

```
((#(6 - + |1| @7 + @23 @24 @28 - - -) "V1/vp_pp"
 (#(6 - + |1| @7 @36049 @23 @24 @28 - - @36050))
 (#(9 - - |2| @36051 @36052 - - @36053 - - @36054)))
```

CF Backbone

The context-free (CF) *backbone* grammar is determined from the object grammar, where rules contain categories identified using atomic names. That is, each atomic name has an associated residue of feature name-value pairs. Given a specified set of features we wish to 'compile' out of the grammar, we effectively determine a rule for each possible combination of these features i.e. so that a CFG (with no residue of features) results if all features of the grammar are compiled out.[2.4] For example, if a rule contains 2 values for 3 features that are unspecified (i.e. have variable values), all of which we wish to compile out, then we create $2^3 = 8$ different run-time rules. Each of these 8 rules now have specific values defined for each of the three features compiled out, and these features no longer need to be unified during parsing.

We determine the CF backbone from the object grammar in two stages. Firstly, we determine the set of disjoint categories in the object grammar. This set covers the whole grammar, where each of the features we wish to compile out have set values in the categories (and thence rules) created. We identify each category with a distinct atomic category name. Secondly, we create a CF rule for each object grammar rule using the atomic category names. Additional refinements are required to the second stage to deal with, for example, coordination and unbounded dependencies, which we do not cover here (see instead, Carroll 1993).

In the resulting CF backbone, nonterminals and terminals in the grammar are then identified using these atomic category names. A table maps these names to the feature-value categories. The grammar for tsg15 consists of 1189 run-time productions, that is, in the compiled CF backbone. The CF backbone consists of 61 distinct categories; 28 terminals, and 33 nonterminals. For example, the CF backbone rule for the grammar rule in Figure 2.12, with `V1-41` and `p2-48` distinct categories, follows:

```
(CFRULE :NAME V1/vp_pp :MOTHER V1-41 :DAUGHTERS (V1-41 P2-48))
V1-41: [N -, V +, BAR 1, PLU @7, MOD @1993, AUX @1994, VFORM @1995,
                          FIN @28, CONJ @2738, SCOLON -, QUOTE @2917]
P2-48: [N -, V -, BAR 2, PSUBCAT @13, PFORM @16, ADJ @18, WH @20,
                          MOD @22, CONJ @2832, SCOLON -, QUOTE @5439]
```

Grammatical Relation (GR) Specifications

Briscoe *et al.* (2006) describe the grammar and the rule-to-rule mapping from local trees to grammatical relations (GRs). The mapping specifies for each grammar rule the semantic head of the rule (head, henceforth), and one or more GRs that should be output (optionally depending on feature values instantiated at parse time). Therefore, the nontrivial task of determining GRs via feature propagation and structural context (over each n-best derivation) is no longer required.

[2.4] Note that this is infeasible given large broad-coverage unification grammars.

For example, the rule in Figure 2.12 identifies the first daughter (`1`) as the head (second line), and that one of five possible GRs is to be output (subsequent lines), depending on the value of the `PSUBCAT` syntactic feature. If the feature has the value `NP`, then the relation is `ncmod` (nonclausal modifier), with slots filled by the heads of the first and second daughters (the `1` and `2` arguments). The resulting *GR specification* is <1, (ncmod _ 1 2)>. That is, each GR specification is defined as: <head, GR>.[2.5]

Figure 2.13 specifies the set of possible GRs defined in the grammar. These GRs are arranged in a subsumption-based hierarchy which illustrates the intent of the grammar writer to differentiate arguments from adjuncts. GRs take the following form: `(relation subtype head dependent initial)` where `relation` specifies the type of relationship between the `head` and `dependent`. The `subtype` slot encodes additional specifications of the relation type for some relations. Finally, the `initial` slot encodes the initial or underlying logical relation of the grammatical subject in constructions such as passive.

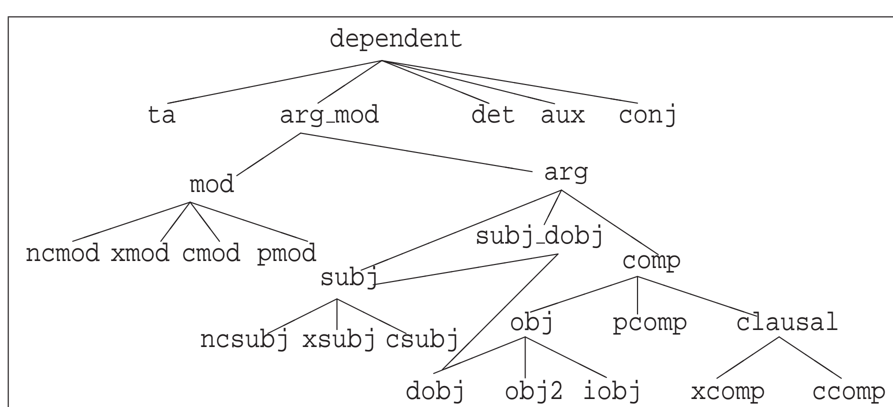

Figure 2.13: The GR subsumption hierarchy

Marked Rules

Of the 1189 productions, 255 are (manually) identified as marked: peripheral rather than core rules of English grammar. These rules are intended to apply only when other rules are not available. For example, there are marked rules to cover heavy NP shifted arguments to verbs where, for example, a short PP argument occurs before an NP: *presented to him the largest case of cigars she had ever seen.*

Optional Grammar Constituents

Many rules include multiple optional constituents (for succinctness of grammar expression). For example, a rule for subject auxiliary inversion with *be* licenses *not* before or after the complement which can also optionally be followed by an adverbial phrase: *is (not) he (not) the abbot (clearly)?*, resulting in $2^3 = 8$ different run-time rules.

[2.5] One or more GRs are defined in the second element of the GR specification. For simplicity, we consider one GR per specification.

2.6.2 Training

LR Table Construction

A generalised LALR(1) (LR, henceforth) table is constructed from the CF backbone grammar utilising the efficient LALR(1) table construction technique, described in §2.3.6. However, this algorithm requires multiple passes over the item sets to propagate the lookaheads correctly between the sets and, as Carroll (1993) reports, was infeasible given the size of RASP's CF backbone grammar. Kristensen & Madsen (1981) describe a modification to this algorithm, applied within RASP, that computes the lookaheads using a single pass over all item sets and caches intermediate results to enable tractable computation of the table.

Action Probabilities

Probabilities are associated with actions in the LR table using the normalisation method of Inui *et al.* (1997) as discussed in §2.5.2. We determine action counts by training on around 5K fully annotated training instances from Susanne (see §1.3.1). During normalisation, we apply Laplace estimation to ensure that we assign a non-zero probability to all actions in the LR table. That is, we assign 'unseen actions' a probability equal to the reciprocal of T, where T is the total frequency of actions plus 1 for each action a in A:

$$T = \sum_{a \in A} (|a| + 1)$$

Each unseen action is in the set A, so that the sum of all action probabilities equals 1.

2.6.3 Parser Application

Processing Components

RASP is based on a pipelined modular architecture in which text is preprocessed by a series of components that perform sentence boundary detection, tokenisation, part of speech (PoS) tagging, named entity recognition and morphological analysis, before being passed to a statistical parser (Briscoe & Carroll, 2002). The PoS tagged tokens are then mapped to terminals of the grammar, so that the input sequence is mapped from raw text to the terminals of the grammar prior to parsing.

Parsing Framework

To parse over an input sequence of grammar terminals, we utilise the GLR framework previously defined in §2.4. That is, we apply a graph-structured stack over the GLR table for the CF backbone. The shift and reduce actions are obtained and applied by considering the atomic categories only. However, after each reduce action we unify the features for the CF rule, effectively augmenting the rule with a test as in Tomita (1987).

Thus, on each reduce action the features and the daughters of the rule are unified. That is, we unify the residue of features not incorporated into the CF backbone grammar. If unification fails then this derivation path also fails. Since unification often fails it is not possible to apply beam or best first search strategies during construction of the parse forest; statistically high scoring paths often end up in unification failure. Hence, the parse forest represents all parses licensed by the grammar.

Efficiency Modifications

Kipps (1989) describes how to turn the *ANCESTORS* function (called for each reduce action) into a table look-up function. Carroll (1993) modifies this algorithm so that the function utilises a *cache* of intermediate results to enable fast look-up of ancestors and these intermediate results are stored in sets of alternative node sequences. Each cache entry keyed on v and k holds the set of all possible node sequences whose start vertex is distance k in the stack from vertex v. The entry is updated when each new analysis is formed during parsing, so that the cache is always up to date. The resulting parser's time complexity is also n^3.

Packing

Carroll (1993) generalises the atomic category packing of Tomita (1987) (described in §2.4.4) to complex feature-based categories following Alshawi (1992). Packing is based on feature structure *subsumption* (Oepen & Carroll, 2000, provide a definition), whereby the most general node represents the set of packed nodes. Miyao & Tsujii (2002) define *feature forests*, an instance of a parse forest in which nodes are sets of property-values rather than, for example, CFG categories. Their *conjunctive* nodes correspond to the node definition we provide, while *disjunctive* nodes represent a set of equivalent conjunctive nodes. In practice, we utilise a single node to represent a set of equivalent nodes. Therefore, packing simply results in a compact data representation by grouping equivalent nodes, regardless of the formalism used to define this structure.

Parse Forest

In view of the multiple root categories, the parse forest is, in practice, considered the set of parse forests, where each dominates analyses spanning the whole sentence with the specified root category. Nodes within the parse forest, and in the resulting derivations are labelled with sets of features (attribute-value pairs). As previously discussed, the LR action table defines the set of possible shift and/or reduce actions applicable given the current state and lookahead. The corresponding action probability assigns a score to each newly derived (sub)analysis, and moreover, to this node in the parse forest. Recall that the parse forest is built within the graph-structured stack, as defined in §2.4.4.

Figure 2.14 shows a simplified parse forest containing three parses generated for the following preprocessed text:

```
I_PPIS1 see+ed_VVD the_AT man_NN1 in_II the_AT park_NN1
```

Each node is uniquely identified by the number in the top left of each square. Edges to circles with numbers indicate that a pointer to a *shared* daughter node is stored. In this case, the number identifies the corresponding node's data structure. For example, node 5 represents the word node for `see+ed_VVD` which is a daughter of nodes 4, 18 and 22.

Parse Selection Model

At parse time, the probability of an action corresponding to a marked production: a 'marked action', is effectively reset to be equal to the minimum of all unseen action probabilities in the table. Due to the marked action probabilities and unification failure, the extant model's probability distributions are deficient. Arguably, the ranking of competing derivations is more important than preserving this property of probability distributions.

Parses are ranked in order from most to least probable, and we determine the n-best derivations, as discussed in §1.1.1. Derivations are found by *unpacking* the parse forest, using a

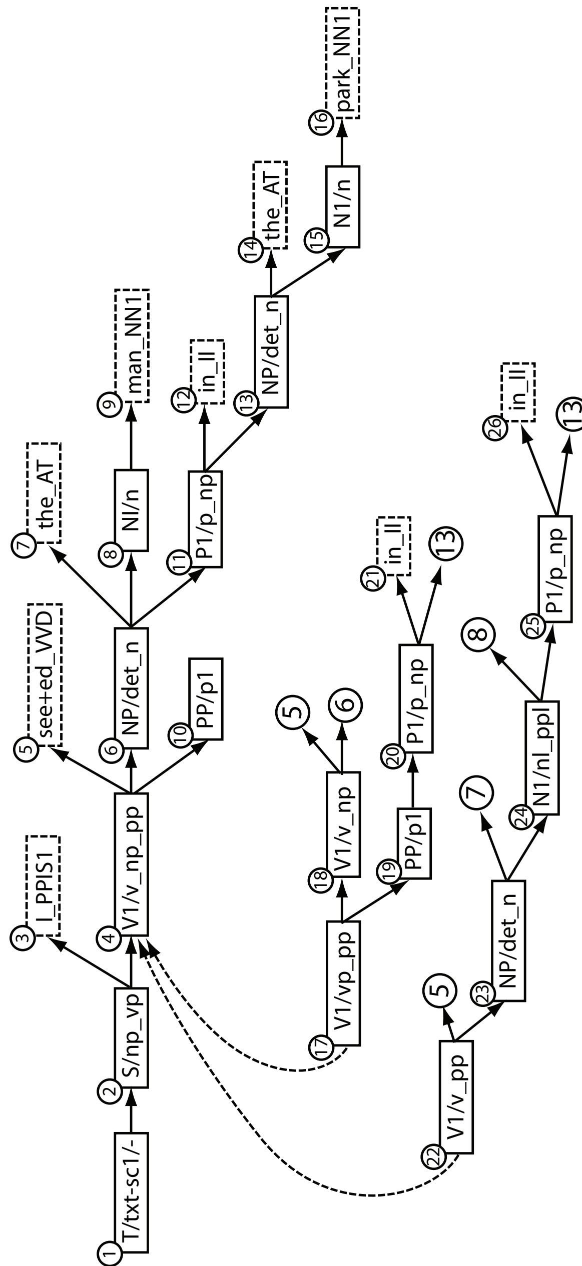

Figure 2.14: Simplified parse forest for *I saw the man in the park*. Each node in the graph represents a node in the parse forest and is shown with the corresponding subanalysis' rule name. Two nodes are packed into the `V1/v_np_pp` node, shown using dotted lines, which results in three alternative parses for the sentence. Edges (with solid lines) illustrate pointers from mother to daughter nodes. Squares shown with dashed lines identify word nodes while solid lines represent tree nodes.

depth-first beam search over complete parse forests (those rooted in top categories of the grammar). The probability of each derivation is the product of all shift/reduce actions that result in the derivation (Briscoe & Carroll 1993, see §2.5.1).

Processing Restrictions

Processing restrictions (time and memory limitations) can be imposed by the user during parsing to enable an efficiency and accuracy trade-off. Further, a word-length limitation may be specified by the user so that only sentences of length less than or equal to this value are processed. As the size of the parse forest scales, in general, exponentially to sentence length, this limitation enables the user to further prioritise efficiency or accuracy.

2.6.4 Output Formats

There are a number of output types available, including syntactic tree[2.6], GRs and robust minimal recursion semantics (RMRS, see Copestake 2003). Each of these is computed from the n-best derivations. It is also possible to specify that a combination of these output formats should be returned.

N-best Derivations

From the parse forest, RASP unpacks the *n-best* derivations.[2.7] Each derivation is represented as an embedded list of pointers to nodes of the parse forest, illustrating the syntactic structure over these nodes. We perform a final unification check across each derivation, as packing is based on subsumption and features may not unify once all combinations of packed nodes are enumerated. If unification fails for an unpacked derivation, then it is removed from the n-best list.

Fragmentary Parse

If the parser is unable to find a full analysis (that is, one rooted in the start category) then the system outputs a *fragmentary* derivation. This is a connected sequence of partial analyses spanning the input by applying a modified shortest paths algorithm (Briscoe & Carroll, 2006). Therefore, given sufficient memory and time, the system is able to produce an analysis for most sentences which lie outside the grammar. If we are unable to create the parse forest within the imposed time or memory limitations, a fragmentary analysis is returned.

Syntactic Tree

A syntactic tree (or tree, for short) consists of labelled-bracketing output in which each set of brackets corresponds to a rule. The contents of each bracket indicates the daughters of the rule, and therefore, the word-span of the phrase itself. The tree is trivial to determine from the derivation, as the list of pointers has the same structure as the syntactic tree. We determine the rule names of the derivation by replacing each pointer to a node with the corresponding original production name (or the word itself for word nodes).

Figure 1.1 shows an example syntactic tree and its corresponding flat labelled-bracketing representation. This flat labelled-bracketing is used to represent syntactic trees output by RASP. For example, Figure 2.15 shows three parses in this format generated for the parse forest shown in Figure 2.14.

[2.6]The term *tree* refers to syntactic trees in this work, while derivation refers to unpacked structures that include full derivation information e.g. attribute-value pairs.

[2.7]The number *n* is specified by the user, and represents the maximum number of parses to be unpacked.

```
(T/txt-sc1/-
 (S/np_vp I_PPIS1
  (V1/v_np see+ed_VVD
   (NP/det_n1 the_AT
    (N1/n1_pp1 (N1/n man_NN1)
     (PP/p1
      (P1/p_np in_II (NP/det_n1 the_AT (N1/n park_NN1)))))))))

(T/txt-sc1/-
 (S/np_vp I_PPIS1
  (V1/v_np_pp see+ed_VVD (NP/det_n1 the_AT (N1/n man_NN1))
   (PP/p1
    (P1/p_np in_II (NP/det_n1 the_AT (N1/n park_NN1)))))))

(T/txt-sc1/-
 (S/np_vp I_PPIS1
  (V1/vp_pp
   (V1/v_np see+ed_VVD (NP/det_n1 the_AT (N1/n man_NN1)))
   (PP/p1
    (P1/p_np in_II (NP/det_n1 the_AT (N1/n park_NN1)))))))
```

Figure 2.15: The n-best syntactic trees output for three parses for the sentence *I saw the man in the park*.

Grammatical Relations (GRs)

The GRs for each derivation are computed from the set of GR specifications at each node, passing the (semantic) head of each subanalysis up to the next higher level in the derivation (beginning from word nodes). GR specifications for nodes are, if required, instantiated based on the features of daughter nodes. These are referred to as *unfilled* until the slots containing numbers are *filled* with the corresponding heads of each daughter node. For example, the grammar rule named `NP/det_n` has the unfilled GR specification <2, (det 2 1)>. Therefore, if a `NP/det_n` local tree has two daughters with heads *the* and *cat* respectively, the resulting filled GR specification is <cat, (det cat the)>. That is, the head of the local tree is *cat* and the GR output is (det cat the). For a derivation, each node has one corresponding head and one or more corresponding GRs (in the filled GR specification). For word nodes, the head is the word itself.

Weighted GRs

The weighted GR output for a sentence consists of the unique set of grammatical relations in all derivations licensed for that sentence, where each GR is weighted based on the probabilities of the derivations in which it occurs. This weight is normalised to fall within the range [0,1] where 1 indicates that all derivations contain the GR.

Example

If we continue the example for the parse forest shown in Figure 2.14, the corresponding shift/reduce probability and instantiated GR specifications for each node are shown in Table 2.9. For example, the `V1/vp_pp` subanalysis (node 17) contains the instantiated GR specification <1, (ncmod _ 1 2)>

since its second daughter has the value NP for its PSUBCAT feature. Note that the head of a word node is considered the word itself.

Node	Word/Rule	Probability	GR SPECIFICATION	
			head	GR
1	T/txt-sc1/–	0.0	1	
2	S/np_vp	-0.5391	2	(ncsubj 2 1 _)
3	I_PPIS1	-0.6763	I_PPIS1	
4	V1/v_np_pp	-0.8728	1	(dobj 1 2),(iobj 1 3)
5	see+ed_VVD	-0.00002	see+ed_VVD	
6	NP/det_n	-1.1568	2	(det 2 1)
7	the_AT	-0.0004	the_AT	
8	N1/n	-1.848	1	
9	man_NN1	0.0	man_NN1	
10	PP/p1	0.0	1	
11	P1/p_np	-0.6565	1	(dobj 1 2)
12	in_II	-0.0134	in_II	
13	NP/det_n	-0.1663	2	(det 2 1)
14	the_AT	-0.0005	the_AT	
15	N1/n	-2.307	1	
16	park_NN1	0.0	park_NN1	
17	V1/vp_pp	0.0	1	(ncmod _ 1 2)
18	V1/v_np	-2.5335	1	(dobj 1 2)
19	PP/p1	0.0	1	
20	P1/p_np	-0.6565	1	(dobj 1 2)
21	in_II	-0.0005	in_II	
22	V1/v_np	-1.1534	1	(dobj 1 2)
23	NP/det_n	-0.1663	2	(det 2 1)
24	N1/n1_pp1	-0.5165	1	(ncmod _ 1 2)
25	P1/p_np	-0.6565	1	(dobj 1 2)
26	in_II	-0.0452	in_II	

Table 2.9: Node (log base 10) probability and instantiated GR specifications for parse forest nodes shown in Figure 2.14.

Figure 2.16 illustrates the n-best GRs and the corresponding (non-normalised and normalised) weighted GRs for this sentence. Weights on the GRs are normalised probabilities representing the weighted proportion of derivations in which the GR occurs. A non-normalised weighting is calculated as the sum of derivation probabilities for derivations that contain the specific GR. We then normalise the weight using the sum of all derivation probabilities. For example, the GR (iobj see+ed in) is in one derivation with probability -8.237, the non-normalised score. The sum of all derivation probabilities is -7.843. Therefore, the normalised probability (and final weight) of the GR is $10^{(-8.237--7.843)} = 0.404265$.[2.8]

[2.8] As we are dealing with log probabilities, summation and subtraction of these probabilities is not straightforward. Multiplication of probabilities X and Y, with log probabilities x and y respectively is determined using the formula $X \times Y = x + y$, division using $X \div Y = x - y$, summation using $X + Y = x + log_{10}(1 + 10^{(y-x)})$ and

N-BEST GRS:
Parse probability: -8.075
(det man_NN1 the_AT)
(det park_NN1 the_AT)
(dobj in_II park_NN1)
(dobj see+ed_VVD man_NN1)
(ncsubj see+ed_VVD I_PPIS1 _)
(ncmod _ man_NN1 in_II)

Parse probability: -8.237
(det man_NN1 the_AT)
(det park_NN1 the_AT)
(dobj in_II park_NN1)
(dobj see+ed_VVD man_NN1)
(ncsubj see+ed_VVD I_PPIS1 _)
(iobj see+ed_VVD in_II)

Parse probability: -9.884
(det man_NN1 the_AT)
(det park_NN1 the_AT)
(dobj in_II park_NN1)
(dobj see+ed_VVD man_NN1)
(ncsubj see+ed_VVD I_PPIS1 _)
(ncmod _ see+ed_VVD in_II)

(NON-NORMALISED) WEIGHTED GRS:
-7.843 (det man_NN1 the_AT)
-7.843 (det park_NN1 the_AT)
-7.843 (dobj in_II park_NN1)
-7.843 (dobj see+ed_VVD man_NN1)
-7.843 (ncsubj see+ed_VVD I_PPIS1 _)
-8.075 (ncmod _ man_NN1 in_II)
-8.237 (iobj see+ed_VVD in_II)
-9.884 (ncmod _ see+ed_VVD in_II)

(NORMALISED) WEIGHTED GRS:
1.0 (det man_NN1 the_AT)
1.0 (det park_NN1 the_AT)
1.0 (dobj in_II park_NN1)
1.0 (dobj see+ed_VVD man_NN1)
1.0 (ncsubj see+ed_VVD I_PPIS1 _)
0.5866 (ncmod _ man_NN1 in_II)
9.0960e-3 (ncmod _ see+ed_VVD in_II)
0.404265 (iobj see+ed_VVD in_II)

Total probability (sum of all parse probabilities): -7.843

Figure 2.16: The n-best GRs, and non-normalised/normalised weighted GRs determined from three parses for the sentence *I saw the man in the park*. Parse probabilities and non-normalised weights are shown as log probabilities as RASP stores all probabilities in log (base 10) form with double float precision.

subtraction using $X - Y = x + log_{10}(1 - 10^{(y-x)})$.

Chapter 3

Part-of-speech Tag Models

We briefly described the preprocessing stages of the extant parser in §2.6.3. These processing modules, which we define in §3.2, include a part-of-speech (PoS) tagger (tagger, henceforth). The tagger maps each token in the input sequence to a set of possible PoS or grammatical categories defined in the tagger's *dictionary* (or *lexicon*). The tagger usually decides on the optimum set of PoS tags given the input sequence, so that only one tag is returned per word. In this case, the tagger is considered to run in *single tag per word (tpw)* mode. However, most taggers may also run in *multiple tpw* mode, where more than one tag is returned for each word, though some of the tag ambiguity present in the dictionary is resolved by the tagger. If a tagger is used as a *front-end* to a parser, then the resulting PoS tags are considered terminals of the parser's grammar. Tag ambiguity unresolved by the tagger is effectively resolved by the parser as the parser selects the most likely derivation, and therefore the corresponding PoS tag sequence, during parsing.

This chapter describes work that aims to optimise the level of tag ambiguity to pass onto the parser, in terms of both parsing efficiency and accuracy. We investigate this aim with respect to RASP's PoS tagger, and define a number of different tag models within RASP's architecture in §3.3. Utilising gold standard tag and GR sets enables comparison of these tag models in terms of both tagging and parser performance in §3.4 and §3.5, respectively. As far as the author is aware, this work is the first to perform such a broad comparison. We first describe related work in §3.1. Much of the work we describe in this chapter appears in Watson (2006).

3.1 Previous Work

This section describes previous work on the integration of PoS taggers into the parsing process. We discuss how PoS taggers may be utilised as front-ends to parsers, that is, as preprocessing modules prior to parsing, in §3.1.1. Next, we describe work that focuses on the choice of tag model, that is, the optimum level of tag ambiguity to pass onto the parser in §3.1.2. Finally, we describe PoS tag models in §3.1.3.

3.1.1 PoS Taggers and Parsers

PoS Tagger Front-end

In §2.3.1, we defined the components of an LR parsing system. We also described natural language parsers; programs that accept a word string as the input sequence and return the corresponding analysis (a parse). The input sequence analysed by the parser consists of terminals of the grammar, so that the parse structure (consisting of nonterminals) is defined and constructed

over these terminals.

Given a CFG, we output a derivation that includes words labelled with PoS tags and non-terminal categories that each span (a subset of) these PoS tags. Depending on the architecture of the parser, we can consider either the words or the PoS tags as the terminals of the grammar. That is, rules of the grammar may include unary rules, where the word is the daughter of the rule and the PoS tag category is the mother of the rule e.g. `Det` → `the`. If we remove these unary rules, and consider PoS tags terminals of the grammar, then we require a preprocessing stage that first maps from the raw text sequence to a PoS tag sequence. In this case, the tagger is considered a *front-end* or preprocessing module to the parser and the input sequence is a set of word-tag pairs e.g. `the_Det`.

Resolving PoS Tag Ambiguity

PoS tagging is the process of mapping from raw text to the PoS *tag sequence*. That is, a method to select one tag per word. A dictionary defines the set of tags for each terminal of the grammar. Initially, the set of tags for each word is considered the set in the dictionary. The tagger then resolves some or all of the tag ambiguity. If it resolves all ambiguity then only one tag per word will remain and the tagger is considered to be in *single tpw* mode.

Conversely if it does not remove any tag ambiguity, we bypass the tagger altogether and allow the parser to resolve all the ambiguity. That is, the parser builds all possible parses relating to each possible combination of tags in the input sequence and selects the most probable parse, and thus, the corresponding tag sequence. In this case, we effectively allow the tag dictionary to define a set of unary grammar rules as discussed previously.

A number of intermediate models may be utilised, whereby we vary the level of tag ambiguity passed onto the parser, so that the tagger and parser are both resolving some of the ambiguity. In this way, the tagger and the parser (or a combination of both) form a PoS tag model. While the parser may act as a PoS tag model, the probability of each tag determined by the tagger, given multiple tpw input to the parser, may be integrated into, thence affect, the probability of each resulting derivation (and corresponding PoS tag sequence).

Affect on Performance

Research investigating the use of PoS taggers as front ends to parsers has, to date, concentrated on whether or not such a preprocessing stage improves parse accuracy and/or efficiency. Compared to the parser, the tagger resolves tag ambiguity efficiently, as parse ambiguity can increase exponentially with tag ambiguity. Thus, most studies agree that efficiency improves with a tagger front-end. For example, Charniak *et al.* (1996) illustrate that using a front-end tagger to resolve all tag ambiguity, which achieves 95.9% tagging accuracy, can significantly improve the efficiency of the parser. Their efficiency metric is the number of edges in their chart parser. They measured tagging accuracy only, and so did not test the impact of these tag models on parsing accuracy.

Researchers have speculated that both speed and accuracy of parsing improves, as the tagger 'filters out' unlikely tags resulting in reduced parse ambiguity. Furthermore, if we reduce tag errors, we reduce the number of parsing errors that result if the parser selects incorrect tags, and therefore, incorrect syntactic analysis over these tags. However, these studies generally employ a 'good' statistical PoS tagger, as taggers now achieve tagging accuracy in the high 90's when trained and tested over the same domain.

Dalrymple (2006) investigates whether we can reduce parse ambiguity if we resolve tag ambiguity. Dalrymple argues that this will only occur if parses can be differentiated based on

their tag sequences. If so, resolving tag ambiguity with a 'perfect tagger' may improve parser performance. Otherwise tag errors introduced by a tagger will be detrimental to parse accuracy and coverage. Dalrymple illustrates that over section 13 of the WSJ (see §1.3.1), parses are differentiated based on their tag sequences for 70.53% of sentences. Given access to a perfect tagger, parse ambiguity is reduced by around 50%.

An increase in tag error rates results in a decrease in parser accuracy and coverage. This decline in performance may out-weigh the benefits of increased efficiency, depending on whether the parsing task prioritises efficiency or accuracy in the parser. For example, Kaplan & King (2003) show that parser coverage (percentage of full parses) falls from 76% to only 62% if they employ a front-end PoS tagger. Dalrymple (2006) suggests that a major source of their tag errors is their mapping from PoS tags to the terminals of their grammar. Furthermore, this error-prone mapping is evident in their results for parser coverage and accuracy over gold-standard PoS tags. In this case, the parser's accuracy still declines, though these gold-standard tags should instead provide an upper bound on the task.

Coverage

Parser coverage is often used to reflect parser accuracy. Although this is not appropriate as increased coverage does not necessarily translate to increased accuracy, especially if the parser's grammar is not well-constrained over the PoS tag terminals. For example, Charniak *et al.* (1996) report that incorrect tags only marginally affect the parser's coverage. Their parser finds a complete parse for all sentences given all possible tags, while only finds a complete parse for 99.2% of sentences over the single tpw input. However, it is unclear whether the accuracy of the parser improves given multiple tpw input as parse ambiguity increases given multiple tpw input. That is, the parser is more likely to select an incorrect sequence of tags (and therefore, an incorrect parse) given all possible tags.

Discussion

Research investigating front-end PoS taggers supports their use if parsing efficiency is paramount. Furthermore, an increase in parser accuracy results if the PoS tagger's tagging errors result in fewer parse errors compared to those due to increased parse ambiguity over multiple tpw input. Thus, the optimal level of tag ambiguity resolved by the parser depends on the given PoS tagger and parser, and furthermore, whether parser efficiency or accuracy is critical in the current parsing task.

For example, Clark & Curran (2004a) illustrate that significant increases in efficiency, accuracy and coverage occur if they integrate a front-end multiple tpw supertagger with their combinatory categorial grammar (CCG) parser. Supertags are lexically rich PoS tags, and specify the local syntactic constraints for a word. They do not fully resolve tag ambiguity in the supertagger, as the supertagger in single tpw mode achieves lower accuracy (low 90's) than PoS taggers with coarser tag sets such as the tagger of Charniak *et al.* (1996).

We do not investigate whether a PoS tagger should be used (readers are referred to Dalrymple 2006 for a recent survey and discussion of this) but instead focus on the choice of tag model. We define a number of tag models in the following section, and provide a broad comparison of these models in terms of a number of performance metrics for both parsing and tagging.

3.1.2 Tag Models

We define a *tag model* as the parsing architecture employed to select the tag sequence or to provide a ranking (probability) for each tag (given more than one tag per word). This may

include utilising the tagger, parser or a mixture of both. Thus the resulting *tag file* can contain any level of tag ambiguity, from single tpw to the full set of tags defined over each word in the tag dictionary. Recall that a *tag sequence* is defined as a sequence of tags (or word-tag pairs) where one tag was selected for each word (token) in the input sequence.

Charniak *et al.* (1996)

Charniak *et al.* (1996) investigate the optimal choice of tag model for a PCFG parser and consider the parser's tagging accuracy, that is, the tag sequence in the top ranked parse. They conclude that the parser, given multiple tpw input, can not significantly improve on the tag accuracy of a (single tpw) PoS tagger. Although they employ only a coarse tag set that consists of only 19 PoS tags. Therefore, these results may not translate for parsers that employ finer grained tag sets (such as the set applied by RASP). Furthermore, if we assume that tag error rates correspond to parser error rates, we incorrectly assume that all tag errors are equally detrimental to parser output. This is not the case however, as for example, a noun mistagged as a verb will have a more adverse effect on parser performance than mistagging the noun as an adjective.

Dalrymple (2006)

Dalrymple (2006) investigates the impact of PoS tags on parse ambiguity; represented by the number of parses licensed by the grammar. By grouping parses via their tag sequences, Dalrymple finds that they can differentiate the majority of parses in terms of their tag sequence since only 30% of sentences had all their parses defined over the same tag sequence. Given the correct tag sequence in these cases, they estimate that parse ambiguity will halve. Dalrymple suggests that the tag sequence that corresponds to the largest number of parses may well be the correct tag sequence given that most parses contain the tag sequence, though she is unable to evaluate this tag selection model without a gold-standard tag set. We consider this tag model in this work, as well as more sophisticated tagging schemes.

Clark & Curran (2004a)

Clark & Curran (2004a) apply a tag model that uses the parser's CCG grammar to decide whether supertags output by the supertagger are acceptable. That is, whether the grammar is able to find a full analysis. A small number of supertags per word are output initially (1.4 tpw), the set of most probable tags. They continue to increase the number of supertags, until either the parser is able to find a complete analysis or the maximum set of tags is considered (those with probability within range of the most probable tag for the word) is considered. This method improves the efficiency, coverage and accuracy of the parser.

3.1.3 HMM PoS Taggers

In this section we describe Hidden Markov Models (HMMs), a common statistical PoS tagging model, as the parser currently utilises a first-order HMM PoS tagger. We also discuss the algorithms that select a PoS tag sequence from the set defined in the tag dictionary. Furthermore, we describe methods that determine this dictionary and train the parameters of a HMM.

Hidden Markov Models (HMMs)

Hidden Markov Models (HMMs) may be viewed as a generalised NFA (see §2.2.1), as tags correspond to states and two types of edges exist. The first type, *transition* edges, links two states in the graph. That is, these edges are the same as the transition edges defined by an NFA.

The corresponding probability of such an edge represents the probability that one tag follows another, or a given set of tags. The *order* of a HMM is the number of preceding tags we consider when we calculate the transition probabilities. For example, a second order HMM determines the probability of a tag based on the previous two tags witnessed. The second type, *lexical* edges, is defined for each state, where we define the probability of a word output by that state i.e. the probability of the word given the tag. Non-zero probabilities for word-tag pairs occur only if this pair occurs in the tag dictionary. We output the tag sequence through the NFA, given the input sequence of words. Although the corresponding state sequence is unknown, resulting in the term 'hidden' in 'Hidden Markov Models'.

Determining Tag Sequences

We determine the tag sequence for an input sequence using the well known Viterbi or Forward-Backward (FBA) algorithms, which are both dynamic programming approaches (readers are referred to Sharman 1990 for a detailed introduction to these algorithms). The Viterbi algorithm selects the tag sequence corresponding to the most likely path through the HMM, where a path's probability is the product of transition and lexical probabilities.

Over an input sequence, the FBA determines the probability of each possible state (tag) in the HMM. That is, non-zero probabilities result for states that are members of one or more complete paths through the HMM. The forward and backward probabilities correspond to the total probability of all paths to and from the state, respectively. The probability of each state is the product of forward and backward probabilities for the state. This probability is the posterior tag probability, that is, the probability of the tag given the word over the given input sequence of words. The most probable tag for each word or all possible tags for the word (with the corresponding posterior tag probabilities) is returned by the tagger. Thresholds can be applied over the posterior probabilities, so that we retain highly-probable tags only in subsequent processing. Furthermore, the posterior tag probabilities can be incorporated into the parser's statistical model.

Training a HMM

If we have a tagged corpus from which to learn the model's parameters (probabilities), then frequency counts can be used to determine both transition and lexical probabilities to produce MLE. The set of tags observed for each word and their frequency are used to create the dictionary. If a hand-tagged corpus is unavailable, we can train a HMM tagger using either the Viterbi or the Baum-Welch algorithm. To perform Baum-Welch training we iteratively update the model's parameters, starting from an initial set of model parameters. During each iteration, we update each tag's probabilities using the forward-backward probabilities of each state (tag). Baum-Welch is designed to converge on a set of parameters that maximise the probability of a training corpus. That is, this training algorithm is a generalised Expectation-Maximisation (EM) algorithm.

The Initial Model's Impact on Performance

Elworthy (1994) and Merialdo (1994) independently define three patterns of Baum-Welch training given different conditions to determine the initial model parameters. Elworthy applies MLE over a hand-tagged corpus to form one of the initial models, from which he refines the model parameters using Baum-Welch over another untagged corpus. The *classical* pattern emerges given a poor initial model, where performance on the test set improves steadily with each training iteration. Two other patterns also result, the *initial* and *early* patterns, where the best model

is either the start model or the one that results from very few training iterations, respectively. In these cases, the initial model either contains sufficient information (e.g. if the model trains over a hand-tagged corpus) or only partial information (e.g. lexical information only), respectively.

Smoothing and Unknown Words

During training, smoothing assigns unseen word-tag pairs a non-zero probability. Furthermore, most taggers also incorporate an unseen word module, so that if a word does not have an entry in the dictionary the tagger is still able to determine the most likely tag or set of tags for each word.

3.2 RASP's Architecture

3.2.1 Processing Stages

RASP is implemented as a series of modules written in C or Common Lisp, which can be pipelined analogously to a series of Unix-style filters. The modules include sentence boundary detection, tokenisation, PoS tagging, NE recognition and morphological analysis. Next, this *preprocessed* text is input to the statistical parser (Briscoe & Carroll, 2002).[3.1] Figure 3.1 illustrates the different processing stages performed during parsing, many of which are optional (e.g. by default NE recognition is not performed). Note that the final two processing stages are both performed by the parser itself.

3.2.2 PoS Tagger

CLAWS Tagset

§2.6.1 describes RASP's grammar, where terminals are defined over PoS tags output by a first order ('bigram') HMM PoS tagger originally implemented by Elworthy (1994). The tagset applied is based on the CLAWS tagset, which is used in Susanne (Sampson 1995, see §1.3.1). The full definition of the tagset is available in this reference, and Jurafsky & Martin (2000)[Appendix C].

In the CLAWS tagset, the first letter encodes the major PoS category and subsequent letters/numbers encode increasingly more minor differences. For example, nouns begin with the letter 'N' and the second letter being a 'P' or 'N' illustrates whether the noun is proper or not, respectively. Tags ending with numbers 1 or 2 illustrate singular or plural versions of the tags. Thus `NP1` and `NP2` are closely related tags, both being proper nouns in singular and plural forms, respectively.

Training

The tagger is trained on 3 million words of text from Susanne, the LOB (Johansson *et al.*, 1986) and (a subset of) the BNC (Aston & Burnard, 1998). The resulting tag dictionary contains just over 50K words. Briscoe & Carroll (2006) make minor modifications to the tagger's dictionary, based on observed parse failures over sections from the WSJ (not including section 23). The frequency counts for each tag are stored in the tag dictionary. Given the training patterns observed by Elworthy, the frequency counts of these fully annotated corpora provide the best tagging accuracy. For example, Figure 3.2 illustrates a number of entries (lines) in the dictionary, where

[3.1] Processing times given throughout, including those in subsequent chapters, do not include these preprocessing stages. We omit these preprocessing overheads as they are negligible compared with those required during parsing.

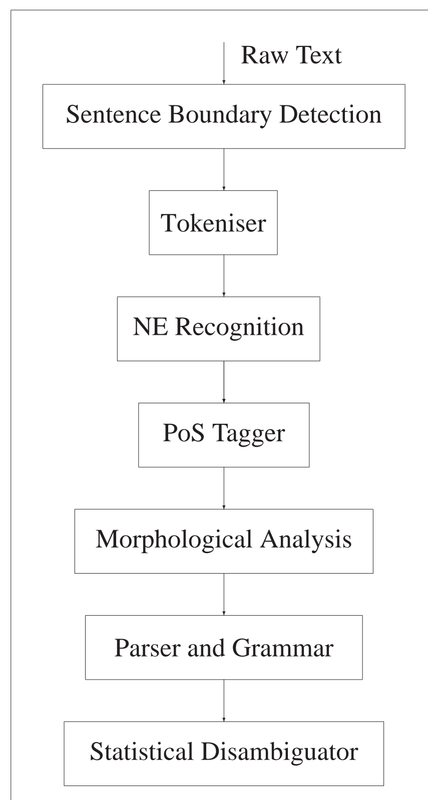

Figure 3.1: RASP processing pipeline.

```
We 2 PPIS2 100 NP1 1
all 3 DB 229 DB2 150 RR 55
walked 2 VVD 46 VVN 13
up 3 II 34 RP 213 VV0 10
the 1 AT 8520
hill 2 NN1 2 NNL1 24
. 1 . 6584
```

Figure 3.2: Example lexical entries in the tag dictionary. Each line in the dictionary corresponds to a word (start of each line) which is followed by the number of PoS tags for the word. The line then contains pairs consisting of a PoS tag and corresponding frequency count.

each specifies the word and corresponding tagset and frequency counts. The sentence *We all walked up the hill* . has the corresponding possible tagset shown for each word in this figure.

Mapping from PoS Tags to Terminal Categories

The full CLAWS tagset contains over 170 PoS and punctuation tags. However, the current grammar only utilises 150 of these, of which around 50 are associated with an identical or subsuming lexical category in the current grammar. That is, the mapping from terminals to PoS tags is many-to-many. For example, the tags `VVD` (past tense form of lexical verb) and `VVG` (-ing form of lexical verb) are both mapped to the terminal with name `V0`. However the verb form values for the terminals (`VFORM` features) are `PAST` and `ING`, respectively. Figure 3.3 illustrates a

```
WORD , : [PUNCT comma].
WORD . : [PUNCT dot].
WORD AT : DT[PLU @x, POSS -, WH -].
WORD CC : CJ[CJTYPE END].
WORD CCB : CJ[CJTYPE END].
WORD DB : N0[NTYPE PART, PLU -, POSS -, WH -, CONJ -].
WORD DB2 : N0[NTYPE PART, PLU +, POSS -, WH -, CONJ -].
WORD NN : N0[NTYPE NORM, PLU @x, POSS -, WH -, CONJ -].
WORD NN1 : N0[NTYPE NORM, PLU -, POSS -, WH -, CONJ -].
WORD PNQO : N2[QUOTE -, SCOLON -, NTYPE PRO, PLU -, POSS -, WH +, CONJ -].
WORD PPIS2 : N2[QUOTE -, SCOLON -, NTYPE PRO, PLU +, POSS -, WH -, CONJ -].
WORD RA : A0[AFORM NONE, ATYPE TEMP, ADV +, CONJ -].
WORD RP : A0[AFORM NONE, ATYPE CAT, ADV +, CONJ -], PT.
WORD VVD : V0[FIN +, AUX -, VFORM PAST, PLU @x, CONJ -].
```

Figure 3.3: Example mapping from PoS tag to terminal category. Each line specifies that an input item (PoS tag) in the grammar is defined, followed by the CLAWS PoS tag and a colon symbol. Following the colon is the metagrammar definition for the given tag (see §2.6.1), where many of the features are fully specified.

number of example mappings between the PoS tags to the terminals of the grammar.

Tagging Modes

The tagger can be run in single tpw (default) or multiple tpw modes, where either the most probable or the set of all possible tags are retained, respectively. As the FBA is implemented in addition to the Viterbi algorithm, the tagger can trade-off precision against recall by returning all but the most improbable tags up to some relative threshold (where tags are ranked according to their posterior probabilities found using the FBA). However, in practice, these thresholds are applied within the parsing module, thus the multiple tpw output of the tagger contains all the tags defined in the tag dictionary. When run in multiple tpw mode, the tagger returns the posterior tag probability of each possible tag.

For example, the sentence *We all walked up the hill .* has the corresponding dictionary entries shown in Figure 3.2. The single and multiple tpw output of the tagger is shown in Figure 3.4. The multiple tpw output shows each tag in the dictionary with the posterior tag probability. The single tpw tag sequence results from selecting the highest scoring tag for each word.

Unknown Word Handling

The tagger incorporates a well-developed statistical unknown word handling module (Piano, 1996; Weischedel *et al.*, 1993) which performs well under most circumstances. However known but rare words often cause problems as tags for all realisations are rarely present. Briscoe *et al.* (2006) describes a series of manually developed rules which they semi-automatically apply to the dictionary to ameliorate this problem, by adding further tags with low counts to rare words. The new tagger has an accuracy of just over 97% on the DepBank part of section 23 of the WSJ (see §1.3.1), which is competitive performance over this (largely out-of-domain) text.

```
We_PPIS2 all_DB2 walked_VVD up_RP the_AT hill_NN1 ._.

We PPIS2:0.999983, NP1:1.73948e-05
all DB:0.168974 DB2:0.803405 RR:0.0276206
walked VVD:0.858121 VVN:0.141879
up II:0.149321 RP:0.850004 VV0:0.000674805
the AT:1
hill NN1:0.607938 NNL1:0.392062
. .:1
```

Figure 3.4: PoS tag output for *We all walked up the hill .* The first line illustrates the single tpw output, while subsequent lines illustrate the multiple tpw output.

3.3 Part-of-speech Tag Models

We tend to consider parser efficiency and accuracy as parsing goals that we must trade-off. However, it is unclear whether errors introduced by a tagger in single tpw mode affect parser accuracy more so than parse selection errors introduced due to increased parse ambiguity. Unless the parser can select PoS tags with greater accuracy, and improve over the parsing performance (of the single tpw tagset) then both efficiency and accuracy improve from the use of a PoS tagger front-end.

In §3.3.1 we describe the various tag models considered in this work. These include the tag models applied by Charniak *et al.* (1996) and suggested by Dalrymple (2006). We contrast these tag models in terms of PoS tag and parser accuracy in subsequent sections, using the gold standard tag and dependency files from DepBank (see §1.3.1). Inclusion of the parser in a tag model assumes that a feed-back loop enables the parser to first select the PoS tag sequence and then (without reparsing) select a parse from the group of parses which contain that tag sequence.

We utilise the first sentence in DepBank to illustrate the different tag files that result for each tag model. This sentence is: *It will also purchase $473 million in assets, and receive $550 million in assistance from the RTC*. Tokenisation results in the following input sequence i.e. with punctuation separated from word forms: *It will also purchase $ 473 million in assets , and receive $ 550 million in assistance from the RTC .*

3.3.1 Part-of-speech Tag Files

Initially we apply the PoS tagger to create PoS tagged files over the raw text files of DepBank. PoS tag files that originate from the tagger alone are named ending with '-TAG'.

SINGLE-TAG

The SINGLE-TAG file contains preprocessed text with the tagger run in forced-choice (single tpw) mode. As a result, a single tag is selected for each token in the sentence and we do not output the associated probability, i.e. each tag has a probability of 1.

ALL-TAG

Similarly, the ALL-TAG file is created by running the tagger in multiple tpw mode. The resulting file contains more than one tag per word i.e. all tags defined for the token in the tag dictionary. When run in multiple-tag mode, the tagger outputs the posterior tag probabilities of each tag as described previously. Figure 3.5 illustrates the SINGLE-TAG and ALL-TAG file

```
It_PPH1 will_VM also_RR purchase_VV0 $_NNU 473_MC million_NNO in_II
assets_NN2 ,_, and_CC receive_VV0 $_NNU 550_MC million_NNO in_II
assistance_NN1 from_II the_AT RTC_NP1 ._.

It It_PPH1:1
will will_NN1:3.41324e-06 will_VM:0.999997
also also_RR:1 also_&FW:1.53286e-09
purchase purchase_VV0:0.959841 purchase_NN1:0.0401588
$ $_NNU:1
473 473_MC:1
million million_NNO:1
in in_RP:8.33575e-308 in_II:1
assets asset+s_VVZ:6.57905e-306 asset+s_NN2:1
, ,_,:1
and and_CC:1
receive receive_VV0:1
$ $_NNU:1
550 550_MC:1
million million_NNO:1
in in_RP:5.443e-308 in_II:1
assistance assistance_NN1:1
from from_RR:0.000113897 from_RG:2.20964e-05 from_II:0.999864
the the_AT:1
RTC RTC_NP1:1
. ._.:1
```

Figure 3.5: PoS tag output for *It will also purchase $ 473 million in assets , and receive $ 550 million in assistance from the RTC* . The first three lines illustrate the single tpw output in the SINGLE-TAG file, while subsequent lines illustrate the multiple tpw output in the ALL-TAG file. Each line in ALL-TAG corresponds to the word at the start of the line, which is followed by each possible tag paired with the corresponding posterior tag probability. Each possible tag is shown in the form: word_tag:probability. Each word in the ALL-TAG file is analysed by the morphological processing module.

contents for the example sentence. In this example, the SINGLE-TAG input is correct, therefore this tag sequence appears in the gold standard tag file.

3.3.2 Thresholding over Tag Probabilities

ALL-TAG contains token-tag pairs with their corresponding posterior tag probability. For efficiency, RASP applies two thresholds over the tag probabilities prior to parsing.[3.2] Firstly, the parser removes all but the most probable tag if this tag has a posterior probability higher than 0.90. Secondly, only tags more than 1 in 50 times as probable as the most probable tag are parsed.

We ‘filter’ the ALL-TAG file to produce the corresponding tag files that result from application of the RASP thresholds, retaining the original posterior tag probabilities. PoS tag files that originate from applying parse system thresholds over the ALL-TAG files alone, are named

[3.2]These thresholds can be specified by the user.

ending with '-SYS'.

DEFAULT-SYS

DEFAULT-SYS results from filtering the full tagset in the ALL-TAG file with the RASP system default tag thresholds. For the example sentence, all ambiguity can be resolved by the application of the system default thresholds. That is, the file contains the same tags as the SINGLE-TAG file, though these are weighted according to the weight of the tag shown in the ALL-TAG file. However, this tag weight has no overall effect on the parse ranking as all parses have the same tag sequence.

MULT-SYS

MULT-SYS results from increasing the system default thresholds to 0.99 and 200, respectively. The tag file contains more tags on average than DEFAULT-SYS due to the lowered thresholds, though still aims to filter out low probability tags. Figure 3.6 illustrates the MULT-SYS tag file contents for the example sentence. Some tag ambiguity remains, as two tags are included for the token *purchase* in the input.

```
It It_PPH1:1
will will_VM:0.999997
also also_RR:1
purchase purchase_VV0:0.959841 purchase_NN1:0.0401588
$ $_NNU:1
473 473_MC:1
million million_NNO:1
in in_II:1
assets assets_NN2:1
, ,_,:1
and and_CC:1
receive receive_VV0:1
$ $_NNU:1
550 550_MC:1
million million_NNO:1
in in_II:1
assistance assistance_NN1:1
from from_II:0.999864
the the_AT:1
RTC RTC_NP1:1
. ._.:1
```

Figure 3.6: MULT-SYS PoS tags for *It will also purchase $ 473 million in assets , and receive $ 550 million in assistance from the RTC .*

3.3.3 Top-ranked Parse Tags

We apply the parser-based tag model considered by Charniak *et al.* (1996), where we select the tag sequence that appears in the top-ranked parse output by the parser. We run RASP over the aforementioned tag files to produce the parser-based files. Time and memory limitations

self-imposed on the system (see §2.6.3) can result in parse time outs, we therefore remove these limitations.

The original posterior tag probabilities affect parse ranking as RASP incorporates these probabilities directly. During parsing, the probability of the word with a given tag is considered the probability of this tag. Thus the probability of a parse is the product of all action probabilities (see §2.6.3) and corresponding posterior tag probabilities. Each tag file contains a single tpw in the same format as that shown for the SINGLE-TAG file. PoS tag files that contain tag sequences corresponding to the parser's top parse are named starting with the original tag file parsed, and ending with '-TOP-PARSE'.

ALL-TAG-TOP-PARSE

The ALL-TAG-TOP-PARSE file contains the top-ranked parse tag sequence when RASP parses the ALL-TAG file.

DEFAULT-SYS-TOP-PARSE

Similarly, the DEFAULT-SYS-TOP-PARSE contains the top-ranked parse tag sequence when DEFAULT-SYS tag file is parsed.

MULT-SYS-TOP-PARSE

Finally, we parse the MULT-SYS tag file to determine the top-ranked parse tag sequence for the MULT-SYS-TOP-PARSE file.

3.3.4 Highest Count Tags

We apply the parser-based tag model mentioned, though not implemented, by Dalrymple (2006). This model selects the tag sequence, whereby each tag selected for a word appears in the highest number of derivations output by the parser.[3.3] The number of derivations containing each tag is normalised by the total number of parses, and this score is used to rerank the tagset for each token. If the system finds a fragmentary parse (see §2.6.4) for a sentence then the system outputs the original tagset for each word.

We consider this reranking over the DEFAULT-SYS file only. This enables feasible processing times as the grammar licenses too many parses given higher levels of tag ambiguity. Ordinarily highly ambiguous sentences are halted due to the time and memory restrictions imposed during parsing, though we remove these as we require tag sequences for all sentences. As we utilise the DEFAULT-SYS file only, and rank tag sequences based on the number of parses, the corresponding tag files start with the name 'DEFAULT-SYS-NUM'. We output the top tag for each word or all tags with the corresponding probability based on the (normalised) number of derivations in which each tag occurs.

DEFAULT-SYS-NUM-TOP

We run RASP over DEFAULT-SYS, and output the top ranked tag sequence where the ranking is based on the proportion of derivations which contain the given tag. This tag file contains a single tag per word, in the same format as shown for the SINGLE-TAG file.

[3.3] As unpacking all derivations is impracticable, we apply the inside-outside algorithm (IOA) to determine these counts/probabilities directly from the parse forest. We describe the IOA fully in the following chapter.

DEFAULT-SYS-NUM-ALL

We output all tags of DEFAULT-SYS, though the tag probabilities are replaced with the new weighting based on the proportion of derivations containing the tag. The tag file contains one or more tags per word, in the same format as shown for the DEFAULT-SYS tag file. However, the corresponding tag weights are updated to the weight determined by this tag model.

3.3.5 Weighted Count Tags

We consider a novel tag model that is effectively a more sophisticated version of the previous tag model based on highest counts. Here, we instead weight tags based on the *weighted* sum of derivations output by the parser. We utilise the corresponding parses' probability in the weighted sum. Therefore, the normalised weight of the tag represents the proportion of parse probability mass containing the tag rather than the proportion of parses.[3.3] Thus tags in higher ranked derivations are considered more likely.

Again, we utilise only the DEFAULT-SYS file to enable feasible processing times over the full data set. The resulting tag files start with the name 'DEFAULT-SYS-WEIGHT'. We output the top ranked tag or all tags (with new probability associated with each) into separate tag files. The following tag files are in the same format as the DEFAULT-SYS-NUM-TOP file and DEFAULT-SYS-NUM-ALL files, respectively.

DEFAULT-SYS-WEIGHT-TOP

We output the top ranked tag sequence where each tag weight represents the proportion of parse probability mass that contains the given tag.

DEFAULT-SYS-WEIGHT-ALL

We output all tags, reweighting each tag with the new probability based on the proportion of parse probability mass.

3.3.6 Gold Standard Tags

We also utilise the gold standard tagset of DepBank (see §1.3.1), to measure the accuracy of the other tag models and to provide an upper bound on parser accuracy. This tag file is referred to as GOLD and contains the single (correct) tag per word, in the same format as shown for the SINGLE-TAG file.

3.3.7 Summary

Table 3.1 summarises the different tag models considered and the corresponding file name for each. The tag model's file name is referred to henceforth.

3.4 Part-of-speech Tagging Performance

This section defines a number of tagging evaluation measures in §3.4.1. These measures are applied to contrast the alternative tag models' performance in §3.4.2.

3.4.1 Evaluation

Standard precision and recall measures are considered, along with a number of other performance metrics which we define here. When determining the correctness of a tag against the gold tag, a few exceptions apply as tags identified as equivalent in the grammar are considered

Tag Setup	Name	Description
Tagger	SINGLE-TAG	Tagger in single tpw mode.
	ALL-TAG	Tagger in multiple tpw mode.
Thresholds	MULT-SYS	Thresholds (0.99, 200) applied to ALL-TAG.
	DEFAULT-SYS	Default thresholds (0.90, 50) applied to ALL-TAG.
Parser	ALL-TAG-TOP-PARSE	Tag sequence in top parse when parsing ALL-TAG.
Thresholds & Parser	DEFAULT-SYS-TOP-PARSE	Tag sequence in top parse when parsing DEFAULT-SYS.
	MULT-SYS-TOP-PARSE	Tag sequence in top parse when parsing MULT-SYS.
	DEFAULT-SYS-NUM-TOP	Most frequently used tag by all parses over DEFAULT-SYS.
	DEFAULT-SYS-NUM-ALL	Normalised counts of tags.
	DEFAULT-SYS-WEIGHT-TOP	Highest scoring tag based upon the sum of probabilities of parses in which tags occur when parsing DEFAULT-SYS.
	DEFAULT-SYS-WEIGHT-ALL	Normalised weighted count of tags.
Manual	GOLD	The gold standard tagset.

Table 3.1: Tag setup descriptions and corresponding file names. The tag setup (first column) defines whether the tagger, thresholds (applied to the posterior tag probabilities output by the tagger) and/or the parser is employed in the tag model. Note that we consider the application of thresholds over posterior tag probabilities a function of the tagger and not the parser (though in practice these thresholds are applied within the parser module). DEFAULT-SYS-NUM- and DEFAULT-SYS-WEIGHT- ALL tag setups are normalised based upon the number of parses and sum of all parse probabilities, respectively.

equivalent for the following metrics. Tags considered equivalent are `&FO` (treated by the grammar as a name) and proper nouns, that is, tags starting with `NP`. Furthermore, all nouns (starting with `N`) and numbers (starting with `M`) are considered equivalent.

Mean Reciprocal Rank (MRR)

MRR is an evaluation metric that can apply if a model produces a list of possible answers ordered by probability of correctness. MRR is used in a number of information extraction and question answering (QA) tasks, and was originally defined in the TREC QA task where the number of answers for each question in the task was set to 5 only.

As several of our tag models produce tag rankings we utilise a generalised version of the TREC QA MRR to measure how well each tag model ranks the tagset for each word. Furthermore, we allow the tagset to be the size specified in the tag dictionary. Other metrics effectively determine boolean values for correctness, that is, reflect whether the top ranked tag is correct. In contrast, the MRR attempts to exemplify systems that are able to rank the correct tag higher. Thus, with identical accuracy, two tag models' MRR differ greatly if one is able to consistently rank the correct tag higher.

Calculation of the MRR is performed using the following equation over a set of tags k_i for each word i in the sentence (the set of words W), where the function *rank* provides the rank of a given tag in the tagset k_i, with correct tag $c_i \in ki$:

$$MRR = \frac{1}{\|W\|} \sum_{i=1}^{\|W\|} \begin{cases} 0 & \text{if } c_i \notin ki, \\ \frac{1}{rank(c_i)} & \text{if } c_i \in k_i. \end{cases}$$

Sentences Affected: *Sent*

We report the proportion of sentences affected by tagging errors in the metric *Sent*, calculated as the percentage of tagged sentences containing at least one tagging error. This metric aims to illustrate the proportion of sentences that are affected by tagging errors as these sentences are more likely to result in parsing errors.

Average Tag Cost: *ATC*

The average tag cost (ATC) is designed to illustrate the average distance between the tag selected and the gold-standard tag, thereby representing the predicted impact on parsing accuracy. In the CLAWS tagset, the first letter encodes major PoS category and subsequent letters/numbers encode more minor differences. Thus, ATC is determined using the average position in which the tag names disagree. If the first letters disagree then this is assumed to be more detrimental than if the last letters or numbers disagree. Therefore, VVD is closer to VVG than to NP1. This measures whether the majority of tag errors are confusions expected to have a detrimental effect on parsing performance.

The distance between two tags is calculated as the reciprocal of 2 to the power of the position in which tag letters or numbers disagree, where the position index begins at 0. In the previous example, the distance is therefore be $\frac{1}{2^2} = 0.25$ and $\frac{1}{2^0} = 1$, respectively. A few exceptions apply to the distance measure: tags identified as equivalent in the grammar (defined previously) have a distance of 0. Confusions between nouns (starting with *N*) and adjectives (starting with *J*) are considered less detrimental and thus have a distance equal to the reciprocal of 2 to the power of the position in which tag letters or number disagree plus one. That is, we define a distance of $\frac{1}{2^{0+1}} = 0.5$.

3.4.2 Results

Table 3.2 illustrates the tagging performance of all eleven tag models measured against the GOLD tag file. The first four rows of this table illustrate the tagging performance of the system's PoS tagger. The following three rows illustrate the performance of the parser's top parse tag model for the three alternative multiple tpw tag models. The remaining four rows illustrate the top parse tag and tag ranking based on the sum of derivations and weighted sum of derivations, respectively.

Tag Setup	Avg tpw[†]	Precision	Recall	MRR	ATC	Sent
SINGLE-TAG	1	97.23	97.23	97.18	0.5757	40.71
DEFAULT-SYS	1.12	88.50	98.79	97.94	-	21.79
MULT-SYS	1.23	80.86	99.42	98.26	-	11.25
ALL-TAG	1.51	65.89	99.78	98.42	-	4.64
DEFAULT-SYS-TOP-PARSE	1	95.38	95.38	95.38	0.6086	59.11
MULT-SYS-TOP-PARSE	1	94.47	94.47	94.41	0.6286	64.46
ALL-TAG-TOP-PARSE	1	93.77	93.77	93.71	0.6496	69.29
DEFAULT-SYS-NUM-TOP	1	92.72	93.86	93.68	0.6325	65.71
DEFAULT-SYS-NUM-ALL	1.12	89.23	98.65	95.99	-	24.11
DEFAULT-SYS-WEIGHT-TOP	1	94.67	95.84	95.66	0.6127	54.82
DEFAULT-SYS-WEIGHT-ALL	1.12	89.23	98.65	97.05	-	24.11

Table 3.2: Tagging Performance.[†]The average tag per word.

PoS Tagger Performance

Accuracy of the tagger in single tpw mode (SINGLE-TAG) on DepBank is good, achieving precision of 97.23%. This precision is higher than might be expected on arbitrary text, as the tagger dictionary has been adapted to the WSJ (described in §3.2.2). Upper bounds on tagging performance are illustrated by the ALL-TAG results, where the only tagging errors are made by the unknown word handling module. Therefore, 4.64% of sentences have at least one incorrect tag due to the presence of 0.22% of words being incorrectly tagged. The proportion of sentences for which at least one tagging error occurs varies dramatically across the four PoS tagger models (from 40.71% to 4.64%). Therefore, if the parser can integrate the multiple tagged data and cope with the additional ambiguity introduced there is limited room for improvement in terms of tag selection.

Parser Tagging Performance

While there is some room for improvement over the PoS tagger, as shown in Table 3.2, none of the alternative parser-based tag models are able to improve on the accuracy of the single tpw output of the tagger (or the ranking of multiple tpw as shown by the MRR score).

As the average number of tags per word increases, the performance of the parser's top parse (the -TOP-PARSE files) declines (reflected in all evaluation measures). However, this will not necessarily translate to poorer parsing performance given that some tags are closer than others (based on the distance metric described). The relatively small decline in the ATC metric, compared to the decline in the PoS tagging accuracy, suggests that a similar drop in parsing accuracy is not expected when moving from single to multiple tpw tagger output.

Highest and Weighted Count Tag Models

The tag model DEFAULT-SYS-NUM-TOP, mentioned by Dalrymple (2006), is the poorest performing tag model. However, RASP's grammar licenses a greater number of derivations (judged over section 13 of the WSJ) than the XLE parser applied in Dalrymple (2006).[3.4] The more sophisticated weighted model (DEFAULT-SYS-WEIGHT-TOP) also performs poorly and furthermore, this model also fails to improve on the ranking of tags (illustrated by the lower MRR score of DEFAULT-SYS-WEIGHT-ALL).

If we consider DEFAULT-SYS-TOP-PARSE the gold standard, and measure the precision of tags in DEFAULT-SYS-NUM-TOP and DEFAULT-SYS-WEIGHT-TOP we find that 94.86% and 99.72% of tags agree respectively. This suggests that the top ranked parse tags tend to occur frequently in the higher ranked derivations but less frequently across all derivations. These findings suggest that it is the statistical model more so than the grammar causing incorrect tag-selection.

Emulating Lower PoS Tagger Performance

In order to emulate performance of the tag models over data with higher levels of unseen words, we determine tagging performance of each model using an initial PoS tagger with artificially reduced performance over DepBank. We reduced the accuracy of the tagging models by around 2%. However, similar results are observed across the tag models and none of the alternative parser-based tag models are able to improve on the accuracy of the PoS tagger.

Furthermore, we test the performance of the tag models over the GDT (see §1.3.1). Whilst RASP is not trained on the GDT, this data represents sentences for which the grammar will have a correct parse. Therefore, this data provides an approximate upper bound on how well the parser can correct over a PoS tagger. However, similar results are also observed over this data set and the initial PoS tagger outperforms all other tag models.

3.5 Parser Performance

While the alternative parser-based tag models are unable to improve on the accuracy of the tagger, this will not necessarily translate to equally detrimental parsing performance. That is, the tagging evaluation measures may not accurately reflect the impact of these models on the parser's performance, given that the parser can recover from certain tag confusions and not others. Therefore, this section discusses the optimal tag model in terms of parser evaluation measures over DepBank. We also contrast the performance achieved using NE markup over DepBank (see §1.3.1). The NE versions of the tag models described previously, are named starting with the original tag file name followed by '-NE'.

3.5.1 Evaluation

We utilise the micro-average and macro-average F_1 measures defined in §1.3.2 against the gold standard (NE) dependency set for DepBank. The proportion of sentences which result in a fragmentary analysis (see §2.6.4) is also reported, along with the time taken to parse the sentences using an Intel Pentium 4 3.2GHz CPU with 1GB of Ram on a 32 bit version of Linux.

[3.4] Dalrymple (2006) reports that over section 13 of the WSJ on average 429 derivations result with a median of 12 derivations. In contrast, RASP finds on average 927K derivations and a median of 128 over this section.

3.5.2 Results

Table 3.3 illustrates the performance of the parser using alternative tag models as front-ends. Ten tag models are shown in the first 20 rows of the table, where each model has two corresponding lines of results. The first line of each pair illustrates the parser's performance over all 560 test sentences. Parsing over the correctly tagged test suite (GOLD) illustrates that a 2.02% increase in F_1 (6.97% relative reduction in error) results from removing the 2.77% of tag errors.

The last two rows of the table illustrate the upper bounds on precision and recall (for all test sentences) when parsing the DEFAULT-SYS tag setup. That is, the precision upper bound is achieved by considering only the GRs resulting from all possible derivations, and the recall upper bound by considering GRs from all possible derivations (Carroll & Briscoe, 2002).

Tag Setup	Micro-average			Macro-average			Frag[†]	Time[‡]
	Prec	Rec	F_1	Prec	Rec	F_1		
SINGLE-TAG	71.06	70.96	71.01	58.08	58.98	58.53	21.25	0:03:50
	73.66	74.94	74.30	62.51	63.45	62.98		
DEFAULT-SYS	71.14	72.21	71.67	58.02	57.64	57.83	12.85	0:05:23
	73.09	74.71	73.89	61.44	60.81	61.12		
MULT-SYS	70.10	71.39	70.74	56.26	57.59	56.92	10.00	0:18:27
	72.07	73.49	72.77	59.66	59.72	59.69		
ALL-TAG	68.42	70.14	69.27	54.61	56.90	55.73	6.96	13:40:32
	70.48	72.24	71.35	60.39	59.00	59.69		
SINGLE-TAG-NE	73.53	69.66	71.54	59.10	58.16	58.63	25.00	0:03:13
	75.66	73.33	74.48	62.88	62.50	62.69		
MULT-SYS-NE	72.54	70.49	71.50	56.96	56.92	56.94	12.68	0:10:57
	74.09	72.62	73.34	62.90	59.16	60.97		
ALL-TAG-NE	71.32	69.30	70.30	55.21	56.13	55.67	9.28	0:45:51
	73.01	71.29	72.14	61.17	58.29	59.70		
DEFAULT-SYS	71.08	72.21	71.64	58.20	57.82	58.01	12.85	0:03:42
-WEIGHT-TOP	72.99	74.69	73.83	61.68	61.04	61.38		
DEFAULT-SYS	67.95	69.11	68.52	53.21	54.85	54.02	12.85	0:03:13
-NUM-TOP	69.59	71.26	70.41	54.90	57.57	56.20		
GOLD	72.94	73.12	73.03	60.53	61.04	60.78	14.46	0:04:39
	74.58	75.70	75.14	64.94	63.91	64.42		
HYBRID	71.59	72.39	71.99	59.02	60.36	59.68	-	-
Precision U/bound	82.25	31.34	45.39	70.22	21.87	33.36	-	4:02:49
Recall U/bound	17.81	87.74	29.60	20.11	82.26	32.32		

Table 3.3: Parser Performance. [†]Frag represents the percentage of fragmentary parses (the percentage of sentences for which full derivations could not be found). [‡]Time is shown in format hours:minutes:seconds.

Parser Performance

If we first compare the performance of the parser for the alternative tag models over all 560 test sentences, it is clear that the most efficient tag model is the tagger in single tpw mode (SINGLE-TAG). The coverage of the parser is increased (the percentage of fragmentary derivations is halved) by using DEFAULT-SYS tag setup. Furthermore, an increase of 0.66% micro-averaged

F_1 results (a relative error reduction of 2.27%), though there is also a similar decrease in macro-averaged F_1. However the macro-average is a less informative metric, as decreased performance can result if incorrect GRs occur in rarer GR types instead.

The increase in parse time required to process DEFAULT-SYS illustrates that there is, as is often the case, a trade-off between accuracy and efficiency. However, the relatively small increase in accuracy will not out-weigh the large decrease in efficiency for most parsing tasks. These results and conclusion agree with those of Charniak *et al.* (1996).

NE Markup

A 0.53% increase in F_1 (1.83% relative reduction in error) results from using gold standard NE mark-up, that is, SINGLE-TAG-NE compared to SINGLE-TAG. Comparing this to the 2.02% increase in F_1 achieved from the gold standard tag set suggests there is more to be gained by concentrating on tag selection than NE recognition. However, this may not be the case over data sets for which a high number of unknown words can be marked as NEs, for example, in biological texts.

Comparison over Full Derivations

In order to compare the impact of the alternative tagging models on parser accuracy, it is also necessary to consider the accuracy for those sentences for which full derivations are found. To compare all tag models across a consistent set of sentences, we consider the sentences for which full (non-fragmentary) derivations result for SINGLE-TAG. That is, we remove 21.25% of sentences that result in fragmentary derivations.[3.5] The accuracy over this set is illustrated in the second row for each tag model in Table 3.3.

The 3.29% increase in F_1 resulting for the SINGLE-TAG tag setup illustrates that a large proportion of the parse errors are introduced by the fragmentary parse output. Further, the margin between the SINGLE-TAG and GOLD tag setups has narrowed to only 0.84% F_1. These results illustrate that tag errors in the SINGLE-TAG file account for a large proportion of the 21.25% resulting fragmentary derivations. This raises an interesting question: can we rely on the grammar to find an analysis if and only if the correct tag sequence is input? A positive response to this question is expected for grammars whose rules (nonterminal categories) are well-constrained over the grammar's terminals (PoS tags).

HYBRID Tag Model

Clark & Curran (2004a) apply a tag selection strategy whereby they assign a small number of supertags per word initially (1.4 tpw) and increase the number of supertags if the parser fails to find an analysis. This is shown to improve the efficiency, coverage and accuracy of the parser. We apply a similar 'dynamic' tag selection model as we observe that single tpw input achieves high accuracy when considering full derivations only, while using multiple tpw input increases the parser's coverage. This model outperforms all others considered herein with respect to parser accuracy. However, efficiency is expected to decrease (from that over single tpw) as this tag model parses each sentence repeatedly until either a full parse is found or we have considered all possible tags for the input sequence.

Supertagging is a harder task than PoS tagging. In single tpw mode, supertaggers achieve much lower accuracy than extant PoS taggers (around 91% compared to 97.23%), suggesting

[3.5] The proportion of fragmentary derivations is around 5% worse than reported by Briscoe & Carroll (2006) as SINGLE-TAG does not include NE mark-up and contains different tokenisation (for example, quotation marks varied significantly).

that resolving the ambiguity during parsing would achieve higher accuracy overall. Moreover, multiple tpw input should be considered for supertaggers as they achieve similar accuracy to PoS taggers in this mode (99.1% compared to 99.78%). Although our dynamic tag model is similar to that of Clark & Curran (2004a), it is a fundamentally different approach due to the higher accuracy of extant PoS taggers over in-domain data. Furthermore, supertaggers are far more sensitive to domain changes as subcategorisation correlates closely with word sense which in turn correlates with the topic/domain.

In order to test this tag selection strategy, we combine the output from the set of sentences that result in full derivations when parsing SINGLE-TAG and the output resulting from parsing DEFAULT-SYS for the remaining sentences.[3.6] The accuracy achieved is illustrated in the row with HYBRID tag setup. This model is the only tag selection model which improves on the accuracy of the parser in terms of both macro- and micro-averaged F_1 (compared over all test sentences). Furthermore, as multiple tpw input is only considered for the fraction of sentences for which a fragmentary parse occurs, the time taken to parse the sentences should also improve; ranging between the time to parse the SINGLE-TAG and DEFAULT-SYS tag setups.

3.6 Discussion

Contrasting the alternative tag models' performance both in terms of tagging and parser performance has supported previous findings. That is, that single tpw input to a parser is sensible given large speed improvements and only a small decrease in accuracy. However, if accuracy is a higher priority than efficiency, then limited tag ambiguity should be passed onto the parser, as accuracy gains are available (0.66% F_1, relative error reduction of 2.27%). Interestingly, the system's default thresholds (optimised on Susanne) for consideration of multiple tags appear near optimal for DepBank, achieving a trade-off between increased parse ambiguity and incorrect PoS tag errors. Significant gains were also made in parser coverage, where using the parser's default thresholds almost halves the number of fragmentary derivations (from 21.25% to 12.85%).

Not one of the tag models based on the parser's output were able to improve on the tagging accuracy achieved by the front-end tagger. That is, the SINGLE-TAG model achieved the highest tagging accuracy. The DEFAULT-SYS-NUM-TOP tag model proved to be the worst performing model in terms of both tagging and parsing performance. Although as previously noted, this may be due to the exponential increase in parse ambiguity given increased tag ambiguity that occurs in RASP. Parsers with more constrained grammars may benefit from the use of this tag model.

The parser was unable to improve on the tagging and parse accuracy achieved by the single tpw PoS tagger. These results may reflect a problem with the integration of the tag probabilities in the statistical model of the parser. We aim to investigate this issue in future work. Again, this may not translate to all grammars, as the number of derivations licensed by our grammar are much higher than achieved by, for example, the XLE grammar (as reported by Dalrymple 2006).

The dynamic (HYBRID) tag model was the only model to improve both macro- and micro-averaged F_1. Here, the known trade-off between parse ambiguity and PoS tag error provides a means to gauge PoS tag error based on parser output. Therefore, when no derivations are found

[3.6] Note that a fully dynamic tag model can be implemented in the system though was not due to time constraints. Instead, output of these two tag files are combined to illustrate the model's performance gain (given only a single iteration of the dynamic model).

the model assumes that a PoS tag error is the cause and increases the number of tags considered. These findings are consistent with those of Clark & Curran (2004b), regarding a dynamic tag model. Though their results were based over a supertagger that achieves lower tagging accuracy in single tpw mode.

Parsing over the gold standard tag set illustrated that a 2.02% increase in F_1 (6.97% relative reduction in error) results. Comparing this to the 0.53% increase in F_1 (1.83% relative reduction in error) which results from using gold standard named-entity (NE) mark-up, suggests there is more to be gained by concentrating on tag selection than NE recognition. However, the conclusions drawn here and by Charniak are based upon high performing PoS taggers that achieve accuracy rates in the high 90's. These conclusions are not expected to translate to parsing systems that employ taggers with lower accuracy. That is, to data with higher levels of unseen words. For example, around 20% of words are 'unseen' in biological texts. In such cases, the use of NE recognition, particularly over the unseen words, is expected to affect parsing performance significantly.

Chapter 4

Efficient Extraction of Weighted GRs

The current parser's output formats, described in §2.6.4, are each determined from the n-best list of derivations. The *weighted GR* output consists of the unique set of GRs across the n-best GR sets, each weighted by the sum of the probabilities of derivations (n-best GR sets) in which it occurs. This weight is normalised (using the sum of all derivation probabilities) to fall within the range [0,1] where 1 indicates that all derivations contain the GR. Hence, the current approach unpacks the n-best derivations from the parse forest, derives the corresponding n-best GR sets and finds the unique set of GRs and corresponding weights.

Carroll & Briscoe (2002) illustrate that increasing the size of the n-best list improves the upper bound on precision/recall in the high precision/recall GR sets, determined by thresholding over the GR weights. Therefore, if practicable, it is preferable to include all possible derivations when calculating weighted GRs. Hence, the extant approach is either (i) inefficient (and for some examples impracticable) if a large number of derivations are licensed by the grammar, or (ii) inaccurate if the number of derivations unpacked (the size of the n-best list) is less than the number licensed by the grammar.

In this chapter we present a novel approach, the EWG algorithm, enabling weighted GRs to be determined directly from the packed parse forest produced by RASP. This approach is a dynamic programming variant of the Inside-Outside algorithm (IOA), which is ideal for this task, enabling exact computation of the GR weights using the inside and outside probabilities determined for nodes in our parse forest. We describe the IOA over *PCFG* parse forests in §4.1. Following, we extend the IOA to apply over LR parse forests, referred to as the IOA_{LR}.

Using the IOA_{LR}, we can determine for each node in the parse forest, the probability of all derivations that include that node. This probability represents the probability of all n-best GR sets that contain the corresponding GR output for the node. Summing over such probabilities for each (unique) GR output (across all nodes in the parse forest) provides the non-normalised weight for the GR. Furthermore, the IOA_{LR} determines the sum of all derivations in the parse forest, which is the normalising factor.

We describe the application of the IOA_{LR} to determine weighted GR output directly from RASP's parse forest in §4.2. A single iteration of the IOA_{LR} is applied to determine the inside and outside probabilities over the parse forest. These probabilities are applied to calculate weights for the GRs which we determine in parallel. Consequently, this approach is referred to as $IOA_{LR}(1)$. However, this solution applies to the extant parser if (i) the parse forest contains only derivations licensed by the grammar. That is, no derivations fail the final unification check we perform over the n-best list of derivations (see §2.6.4). In addition, the solution requires that

(ii) each node in the parse forest is assigned a single semantic (lexical) head. This allows us to consider the product of inside and outside probabilities for each node as the corresponding probability of the node's lexical head.

EWG, defined in §4.3, ensures that (i) holds for all input by altering the local ambiguity packing operation. EWG does not ensure that (ii) holds, while related work has, instead we modify the $\text{IOA}_{LR}(1)$ so that we allow each node in the parse forest to be assigned *multiple* inside and outside probabilities, one for each possible lexical head. Consequently, EWG improves on previous work which either loses efficiency by unpacking the parse forest before extracting weighted GRs, or places extra constraints on which nodes can be packed to ensure that each node specifies a single lexical head, leading to less compact parse forests.

Our experiments, described in §4.4, demonstrate substantial increases in parser accuracy and throughput for weighted GR output. Finally, in §4.5, we apply a parse selection strategy defined by Clark & Curran (2004b) that utilises the EWG algorithm and achieve 3.01% relative reduction in error. Furthermore, the GR set output by this approach is a *consistent set*. In contrast, the high precision GR sets defined in Carroll & Briscoe (2002) are neither consistent nor coherent. [4.1] Much of the work we describe in this chapter appears in Watson *et al.* (2005).

4.1 Inside-Outside Algorithm (IOA)

We first describe the development and properties of the IOA in §4.1.1. We then define the IOA and its standard (iterative) application to PCFG training in §4.1.2. Finally, in §4.1.3 we illustrate the simple extension to apply the IOA to the parse forest derived from an LR parser and to the extant parser's forest.

4.1.1 Background

The Inside-Outside algorithm (IOA) was introduced by Baker (1979), as a generalisation of the HMM's Baum-Welch estimation methods (see §3.1.3), to enable re-estimation of parameters in PCFGs over raw (unannotated) text. Inside and outside probabilities are analogous to the forward and backward probabilities of the Baum-Welch algorithm. The IOA was reviewed by Lari & Young (1990), who extended it to an iterative training method for PCFGs that allows the grammar to be inferred from a training corpus of unrestricted size.

Relationship to EM

The IOA is a variant of the Expectation-Maximisation (EM) algorithm, in which the basic assumption is that a 'good' grammar is one for which training sentences are likely to occur. That is, the IOA as described by Lari & Young (1990), is used to find the MLE over training corpora that are not annotated. Prescher (2001) formally proves that Inside-outside estimation is a dynamic-programming variant of EM, and therefore, inherits the good convergence behaviour of EM. That is, the IOA converges on a set of parameter estimates that maximise the probability of the training corpus.

Local Maxima and Convergence Patterns

The IOA is not without problems. For example, Charniak (1994) illustrates that the algorithm converges to different local maxima given different initial estimates (randomly selected) for

[4.1] A *consistent set* of GRs is one in which each word is the dependent of only one other word and a *complete set* is one in which every word is listed as the dependent of another word.

model parameters. Further, Elworthy (1994) found three general patterns for the related Baum-Welch algorithm over HMM PoS taggers, as described previously in §3.1.3, in which the algorithm did not necessarily improve given a 'good' initial model.

Discussion

We illustrate IOA (EM) training over the extant LR parser in the next chapter. Although the convergence properties of IOA are not of concern in this chapter, as EWG simply applies a single iteration of a variant of IOA to determine the probability of all derivations containing each GR in the parse forest. That is, we utilise the dynamic programming properties of the algorithm to efficiently compute weights of elements in the data structure. Several parse selection and training strategies employ similar dynamic programming approaches, which are necessary for efficient parser training and application, as unpacking the entire parse forest is unfeasible for broad coverage natural language parsers. For example, Miyao & Tsujii (2002) describe a variant of the IOA for training a log-linear parsing model from packed feature forests.

4.1.2 The Standard Algorithm

Much of the theory described in this section, regarding the 'standard' IOA application to PCFG training, has been adapted from Lari & Young (1990). We discuss the application over a *PCFG* parse forest. That is, to a parse forest in which nodes result from applying a rule of the grammar, thence each node represents the mother category i of the rule only. Each subanalysis for a node N_i results due to the application of a rule of the grammar of form $i \rightarrow jk$ or $i \rightarrow j$. That is, we assume that the PCFG is in Chomsky Normal Form (CNF).

Overview

We previously described forward and backward probabilities in a HMM (which may be considered a generalised NFA) in §3.1.3. The probability of a state in the HMM is considered the product of forward and backward probabilities for the state. That is, the total probability of all complete paths to and from the state, respectively. Similarly, we consider the probability of nodes in a PCFG parse forest as the product of the inside and outside probabilities (the IO probability) for the node N_i. This probability corresponds to the sum of derivation probabilities over all derivations that contain the NT category i over the node's word span. For example, for the grammar rule `NP/det_n` over the input *the man*, the corresponding node's IO probability is equal to the probability of all derivations which include the `NP/det_n` category over this subset of the input.

We consider the sum of all such IO probabilities for each NT category i, for each possible word span for nodes N_i in each sentence in the training corpus, to represent the expected frequency of the nonterminal. Thus we effectively re-estimate the probability of the rule using relative frequency counts, that is, for production $i \rightarrow jk$, the probability of the rule is determined using:

$$P(i \rightarrow ij) = \frac{freq(i \rightarrow ij)}{\sum_{j,k} freq(i \rightarrow jk)} \tag{4.1}$$

Thus the IOA proceeds by iteratively re-estimating the probability for each rule in the grammar where each iteration involves determining such relative frequency counts over the entire training corpus. This process results in converging the probabilities of each rule so that the probability of the training corpus is maximised. Convergence occurs in practice, as rule rewrites that occur more frequently for each NT category are at each iteration assigned a higher proportion of

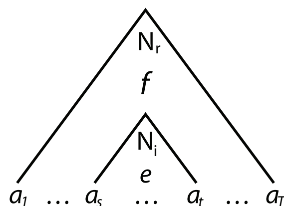

Figure 4.1: The inside (e) and outside (f) regions for node N_i.

the probability mass for the category. Thus, the set of sentences that contain these rules each have a higher probability so that the total probability of these sentences in the corpus increases. However, to maximise the probability of the entire training corpus, we must ensure that low frequency rules are still assigned some of the probability mass. The IOA converges on a maximum that assigns more weight to frequently occurring rules yet still determines all training sentences to occur with 'high' probability.

The inside and outside probabilities for nodes are defined herein, which are applied to determine these expected frequency counts for each rule. We also define the re-estimation equations that effectively apply Equation 4.1, so that we iteratively converge on a (locally) maximal solution. That is, to perform EM over PCFG.

Inside Probability

We defined a CFG grammar G, in §2.1, as a tuple $\{NT, \Sigma, P, R\}$. The NT and Σ elements represent the set of nonterminal and terminal symbols of the grammar, respectively. The element P represents the set of productions (rules), while R represents the nonterminal category that is considered the top grammar category.

The probability of a subanalysis is calculated as the probability of the rule applied (that created the subanalysis), multiplied by the product over all daughter node probabilities. The inside probability of a node represents the probability of all subanalyses for the node, calculated by summing over their corresponding probability. Conversely, the outside probability represents the probability of all analyses which include one of the node's subanalyses.

Given an input sequence of terminals of the grammar $\{a_1, ...a_T\}$, we denote the inside and outside probabilities for a node N_i, that spans input items a_s to a_t inclusively, as $e(s,t,N_i)$ and $f(s,t,N_i)$, respectively. Figure 4.1 illustrates the corresponding nodes in the parse forest used when calculating the inside and outside probabilities for N_i. Nonleaf nodes in the figure represent NT categories, and N_r is the root node whose category r is in the set R. Leaf nodes represent Σ categories of the grammar, that is, the input sequence $\{a_1, ...a_T\}$.

The inside probability $e(s,t,N_i)$ represents the probability of subanalyses that are rooted with mother category i for this sentence over the word span s to t. Each production is of the form $i \rightarrow jk$ where each set of daughter nodes N_j and N_k span from a_s to a_r and a_{r+1} to a_t, respectively. Figure 4.2 illustrates this structure for node N_i. The inside probability $e(s,t,N_i)$ for a given sentence is calculated using:

$$e(s,t,N_i) = \sum_{j,k} [P(i \rightarrow jk) \sum_{r=s}^{t-1} e(s,r,N_j)e(r+1,t,N_k)] \tag{4.2}$$

We calculate the inside probabilities bottom-up. For a word node, the inside probability is

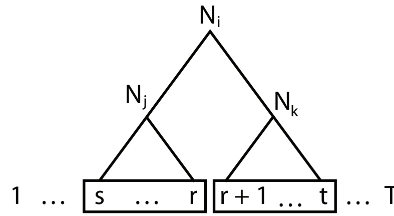

Figure 4.2: Calculation of inside probabilities for node N_i.

either equal to the PoS tag probability (if PoS tag terminals have associated posterior tag probabilities) or considered to be 1. That is, for the input item a_s for word number s, $e(s,s,N_{a_s}) = 1$.

As a result, over a single derivation, the inside probability of each node corresponds to the product of all CFG rules that are applied to create the subanalysis. That is, the top category's inside probability is the derivation's probability. Over the parse forest, the inside probability of the root node (that is, the probability for sentence q, $P_q = e(1,T_q,N_{R_q})$) corresponds to the summation over the probabilities of all derivations for the sentence.

Outside Probability

The outside probability for a node N_i, as shown in Figure 4.1, is calculated using all the nodes for which the node is a daughter (subanalysis). This calculation includes the inside probability of the other daughter nodes of which N_i is a member. That is, category i could appear in two settings: $j \rightarrow ik$ or $j \rightarrow ki$, as shown in Figure 4.3.

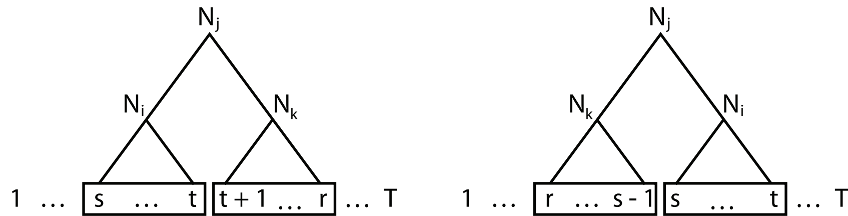

Figure 4.3: Calculation of outside probabilities for node N_i.

The outside probability of N_i, $f(s,t,N_i)$, represents the probability of all possible analyses that include the node N_i and span from a_1 to a_{s-1} and a_{t+1} to a_T. We calculate the outside probability of N_i, using the outside probability of the mother node (N_j) multiplied by the product of inside probabilities of the daughters other than N_i i.e. N_k. We perform a summation over this probability in each instance where N_i is a daughter of a node, i.e. for each possible N_j and N_k. The outside probability $f(s,t,N_i)$ for a given sentence is calculated using:

$$
\begin{aligned}
f(s,t,N_i) \;=\; & \sum_{j,k}[f(s,r,N_j)P(j \rightarrow ik) \sum_{r=t+1}^{T} e(t+1,r,N_k)] \\
& + \sum_{j,k}[f(r,t,N_j)P(j \rightarrow ki) \sum_{r=1}^{s-1} e(r,s-1,N_k)] \qquad (4.3)
\end{aligned}
$$

We calculate the outside probabilities top-down, assigning the outside probability of the root node N_r to be 1.

Re-estimation Equations

We rearrange the standard inside Equation 4.2, splitting the equation into two separate equations. The first equation specifies the rule used, that is, the particular NT categories for j and k. We perform the summation over each possible set of categories for j and k in the second equation:

$$e(s,t,N_i,N_j,N_k) = P(i \rightarrow jk) \sum_{r=s}^{t-1} e(s,r,N_j) e(r+1,t,N_k) \tag{4.4}$$

$$e(s,t,N_i) = \sum_{j,k} e(s,t,N_i,N_j,N_k) \tag{4.5}$$

The product of inside and outside probabilities $e(s,t,N_i)$ and $f(s,t,N_i)$ represents the sum of all probabilities for derivations in which the mother category i is utilised over the span a_s to a_t. If we sum over all possible values for s and t for the sentence, then we determine the sum of all probabilities for derivations in which the mother category i appears. If we normalise by the probability of all derivations (P_q) then we determine the proportion of the probability mass within the parse forest for sentence q that utilises the mother category i. Within the IOA, this quantity falls in range [0,1], and is effectively considered the frequency of the mother category i for the sentence:

$$freq_q(N_i) = \frac{1}{P_q} \sum_{s,t} e_q(s,t,N_i) f_q(s,t,N_i)$$

Similarly, the product of the inside and outside probabilities, $e(s,t,N_i,N_j,N_k)$ and $f(s,t,N_i)$, represents the probability of all derivations in which the rule $i \rightarrow jk$ is utilised over the span a_s to a_t. Again, if we sum over the values for s and t, and normalise using P_q, then we determine the weighted proportion of derivations in which the rule $i \rightarrow jk$ occurs:

$$freq_q(N_i,N_j,N_k) = \frac{1}{P_q} \sum_{s,t} e_q(s,t,N_i,N_j,N_k) f_q(s,t,N_i) \tag{4.6}$$

Dividing this proportion by the proportion determined for mother category i calculates the relative weighted frequency (i.e. probability) of expanding i using the rule $i \rightarrow jk$ for sentence q:

$$P_q(N_i,N_j,N_k) = \frac{freq_q(N_i,N_j,N_k)}{freq_q(N_i)} \tag{4.7}$$

Extending this equation to apply over all sentences for a corpus simply involves summing over each sentence q in corpus Q. The following equation represents the relative frequency count Equation 4.1, and is the re-estimation equation applied during each IOA iteration, to determine the new probability of each rule $i \rightarrow jk$ of the grammar:

$$P(i \rightarrow jk) = P(N_i,N_j,N_k) = \frac{\sum_{q \in Q} freq_q(N_i,N_j,N_k)}{\sum_{q \in Q} freq_q(N_i)} \tag{4.8}$$

4.1.3 Extension to LR Parsers

We described the IOA for a PCFG parse forest. However, each node in a PCFG parse forest corresponds to a particular mother category of the grammar. As described in §2.3.6, an LR parser encodes additional context over the underlying CFG. In view of this additional context, we apply the re-estimation equations for each action in the LR table, rather than for each CFG production. Therefore this is not a 'standard' IOA implementation, and consequently, is referred to as the IOA_{LR} instead.

Calculating Inside and Outside Probabilities

The extension of the IOA to LR parsers is relatively trivial, as we determine the inside and outside probabilities of each node in an LR parse forest as we likewise determine for each node in the PCFG parse forest. Each frequency calculation now relates to the parse action in the LR table, rather than to the CFG rule applied. That is, we utilise the corresponding probability of the action for node N_i, $P(a[i])$ rather than that of the corresponding rule $P(i \rightarrow jk)$. The product of the inside and outside probabilities (the IO probability) for each action corresponds to the sum of probabilities for all derivations in the LR parse forest that result due to the LR parse action. For a word node, the (inside) probability is equal to the probability of the shift action that results in creation of the word node and (optionally) the posterior PoS tag probability.

Action Counts and Normalisation

In the LR parse forest, each node N_i represents an LR state that spans a subset of the input sequence. The set of actions for N_i, that result in a subanalysis for N_i (and moving to the LR state), is considered rather than the set of rules applied. These actions are usually defined in different cells of the LR table, as each subanalysis results from a reduce action over different daughter nodes (representing varying left-context, thus states, in the LR parser).

We determine the frequency of each action using the normalised IO probability of the action. That is, we apply Equation 4.6 (as described in the following section). However, we do not apply the normalisation factor in Equation 4.7. Instead, we normalise based on the set of competing actions for the current state and/or lookahead item rather than the node's mother category or LR state. For each set of competing actions in the LR table, we normalise each action in the set by the sum over these action frequencies. Therefore, the frequency of each action calculated using Equation 4.6 is considered the action count for each action and we normalise over these counts as we would otherwise over a fully annotated corpus for an LR parser (see the normalisation methods defined in §2.5.2).

Extension to the Extant Parser

In addition to the minor modifications made to the algorithm to apply over LR parse actions (rather than PCFG rules), there are a number of practical differences in the application of the algorithm described to the parse forest data structure built by the extant parser.

Nodes in the parse forest are efficiently represented using sub-tree sharing and packing (see §2.4.4). In keeping with previous notation, a set of nodes in the parse forest will be referred to using uppercase N. The representative node for the set N_i is the node n_i, where the number i uniquely identifies a node (set) in the parse forest. Each node $n_c \in N_i$ represents a particular action. That is, a single subanalysis for the node set N_i. The representative node n_i may contain a set of packed nodes n_p, where $n_p = \text{PACKED}(n_i)$. Thus, $N_i = \{n_i | PACKED(n_i)\}$. For example, see Figure 2.14, which illustrates a parse forest produced by the extant parser. There are 26 individual nodes, two of which are packed in another, resulting in 24 node sets. The node n_4

for `V1/v_np_pp` represents the fourth node set N_4, which includes the node itself and the packed nodes n_{17} (`V1/vp_pp`) and n_{22} (`V1/v_np`).

Each node $n_c \in N_i$ has a set span (set values for s and t), and corresponds to a single parse action. Similarly, the nodes' daughter node sets $DN_l \in D(n_c)$ do not vary, that is, each has a set span and mother category. Further, our grammar is not in CNF and more than one daughter may be specified for a grammar rule. As a result, in Equation 4.4, the values j, k and r are effectively set. We determine the inside probability of the node using a simplified version of this equation:

$$e(n_c) = P(a[c]) \prod_{DN_l \in D(n_c)} e(DN_l) \tag{4.9}$$

Note that we no longer specify the span of the node in this equation, as this is already set for each node in the parse forest and each node n_c represents a specific data structure in the parse forest. We use $a[c]$ to represent the action for node n_c that applies a rule of form $A \rightarrow \alpha$, where the set of daughter categories α corresponds to the set of categories in each of the daughter node sets $D(n_c)$.

As packed nodes represent alternative subanalyses for the node, we apply the summation for the inside equation. That is, we apply Equation 4.5 over the set of nodes in N_i:

$$e(N_i) = \sum_{n_c \in N_i} e(n_c) \tag{4.10}$$

We store the inside probability for the node itself n_c and utilise this probability to calculate the node's IO probability. However we return the inside probability for the node n_i as that of N_i.

Simplifying the outside equations, given the set data structure of the parse forest, is performed in a similar fashion. We simplify Equation 4.3, as the value r is set, as are the daughters for each mother node n_j of N_i, to:

$$\forall n_c \in N_i, f(n_c) = f(N_i) = \sum_{n_j, N_i \in D(n_j)} f(n_j) P(a[j]) \prod_{N_k \in D(n_j), N_k \neq N_i} e(N_k) \tag{4.11}$$

Different nodes in the parse forest can result from the same parse action. To determine the frequency count for each action a_d for sentence q, we sum over the IO probability for each node that results due to application of the action. That is, we simplify Equation 4.6 to:

$$freq_q(a_d) = \frac{1}{P_q} \sum_{n_c, a[c] = a_d} e_q(n_c) f_q(n_c) \tag{4.12}$$

Again, we determine this frequency across each sentence q in the training corpus Q. However, we determine the nominator in Equation 4.8 only:

$$freq(a_d) = \sum_{q \in Q} freq_q(a_d)$$

The denominator for Equation 4.8 is instead calculated by summing over the frequency counts for competing actions in the LR table. That is, we normalise the frequency counts for actions using the extant normalisation method described in §2.6.2.

4.2 Extracting Grammatical Relations

In this section, we show how to obviate the need to trade off efficiency and accuracy by extracting weighted GRs directly from the parse forest using a dynamic programming variant of the IOA_{LR}; the $IOA_{LR}(1)$. This approach, extended in §4.3 to handle all parse forests output by the extant parser, enables efficient calculation of weighted GRs over *all derivations* and substantially improves the throughput and memory usage of the parser.

Our approach removes the intermediate processing stages that unpack the n-best derivations and determine the corresponding set of GRs for each derivation. Instead, we show how to extract the weighted GR output directly from the parse forest. We assume for now that a single lexical head is specified per node. In this case, we consider the IO probability of the node as the IO probability of the node's GR specification. This probability forms part of the non-normalised score for the GR.

In §4.2.2, we describe the $IOA_{LR}(1)$, which determines weighted GRs directly from the extant parse forest. We apply Equation 4.12, though over the unique GR g_d rather than for the action a_d for each node. Here, $freq_q(g_d)$ is the final weight for unique GR g_d in the weighted GR output, where $g[c]$ represents the GR output for node n_c:

$$freq_q(g_d) = \frac{1}{P_q} \sum_{n_c, g[c]=g_d} e_q(n_c) f_q(n_c) \tag{4.13}$$

We continue the example over the sentence *I saw a man in the park*, the parse forest for which is shown in Figure 2.14. The probability of shift/reduce actions and instantiated GR specifications for each node are shown in Table 2.9. Figure 2.16 illustrates the corresponding n-best GR sets and weighted GR output formats for this example.

In practice, more than one lexical head can result for each node in the parse forest. We illustrate this by example in §4.2.3, artificially modifying the GR specifications in our continued example so that more than one lexical head results for a node in the parse forest. We illustrate that the $IOA_{LR}(1)$ is unable to extract the corresponding weighted GR output from the resulting parse forest.

Since the parser is unification-based, we first discuss in §4.2.1, modifications to the parsing algorithm so that local ambiguity packing is based on feature structure *equality* rather than subsumption. This ensures that only derivations licensed by the grammar are represented within the parse forest.

4.2.1 Modification to Local Ambiguity Packing

We described the extant parser's packing method in §2.6.3, where packing is based on feature structure subsumption. In this section, we discuss the problem with this packing definition in relation to applying dynamic-programming approaches to the parse forest. We also describe how to modify the packing operation to enable weighted GRs to be extracted directly from the parse forest.

Global Consistency for Subsumption-based Packing

Oepen & Carroll (2000) note that when using subsumption-based packing with a unification-based grammar, the parse forest may implicitly represent some derivations that are not actually licensed by the grammar. These derivations have values for one or more features that are locally but not globally consistent. This is not a problem when computing GRs from derivations that

have already been unpacked. In this case, the relevant unifications are checked during the unpacking process, and unification failure causes the affected derivations to be filtered out. Unification fails for at least one packed tree in approximately 10% of the sentences in the DepBank test suite (see §1.3.1). However, such inconsistent derivations are a problem for any approach to probability computation over the parse forest that is based on the IOA. For EWG, we therefore modify the parsing algorithm so that packing is based on feature structure *equality* rather than subsumption.

Packing Operations

Oepen and Carroll give definitions and implementation details for subsumption and equality operations, which we adopt. In the experiments below, we refer to versions of the extant parser with subsumption and equality based packing as SUB-PACKING and EQ-PACKING respectively.

When packing based on subsumption, node $n_{\mathtt{B}}$ is packed into node $n_{\mathtt{A}}$ based on the truth value returned by the following definition of subsumption:

> Given two nodes $n_{\mathtt{A}}$ and $n_{\mathtt{B}}$ with (unification-based) feature lists Θ_A and Θ_B, then $n_{\mathtt{A}}$ subsumes $n_{\mathtt{B}}$ if and only if:
>
> 1. the category type of A and B, specified by the first elements in Θ_A and Θ_B, are equal; and
> 2. each remaining corresponding element of Θ_A and Θ_B, θ_A and θ_B respectively, conform to one of the following:
>   (a) θ_A and θ_B both contain values and these values are equal or
>   (b) θ_A is unspecified

The definition of equality between nodes is similar to the above definition except for a minor change to part 2b, requiring *both* θ_A *and* θ_B to be unspecified.

4.2.2 Extracting Grammatical Relations

Unfilled GR Specifications

The extant parser's grammar, described in §2.6.1, encodes a set of possible GR specifications for each rule of the grammar. The particular GR specification that applies for the resulting node n_c in the parse forest may depend on the feature values of the nodes daughters $D(n_c)$. That is, the grammar defines a rule-to-rule mapping from local trees to GR specifications that are optionally instantiated during parse time so that one GR specification, in the form <head, GR> is specified for each node n_c in the parse forest. The head of the GR specification defines the semantic head (head, henceforth) for this node, while the GR is the GR output for the node.

The GR specifications for each node in our example sentence are shown in Table 2.9. These are *unfilled* (see §2.6.4), as the GR slots are specified using the number of the daughter whose head should be used. Node 13 for `NP/det_n` spans over the subset of the input *the park* and has the unfilled GR specification <2, (det 2 1)>. This GR specification defines that the second daughter's head is the node's head, and the GR output by the node is a det(erminer) where the `head` and `dependent` slots are to be filled by the heads of the second and first daughters, respectively (the `2` and `1` arguments).

Filled GR Specifications

GR specifications are referred to as unfilled until the slots containing numbers are *filled* with the corresponding head of each daughter node. We consider the head of each word node as the word itself. For example, the resulting filled GR specification for node 13 is <park, (det park the)>, i.e. the head of the node is *park* and the GR output is (det park the).

Over a single derivation, the corresponding set of GRs is computed from the instantiated GR specifications at each node in the derivation, passing the head determined for each node (for the filled GR specification) upwards from daughter to mother nodes in the derivation. For example, the corresponding n-best GR sets for each derivation output for our example sentence are shown in Figure 2.16.

Processing Stages

Three processing stages are required to determine weighted GRs over the parse forest. That is, to apply the $\text{IOA}_{LR}(1)$. We calculate (1) filled GR specifications and corresponding inside probabilities, (2) outside (and non-normalised) probabilities of a unique GR set, and (3) normalised probabilities (that is, the final weights) of the weighted GR set. Note that the inside and outside probabilities in the first two stages are effectively determined using a single iteration of the IOA_{LR}, though we do not update our model's parameters.

The first two processing stages are covered in detail in the following sections. The final stage normalises the probabilities of the unique GR set by dividing each weight by the sum of all the derivation probabilities. That is, by dividing with the root-node's inside probability (P_q for sentence q). Stage (3) can instead be performed during stage (2), so that we incrementally determine the normalised probabilities of each unique GR, as we determine this normalising factor during stage (1).

Inside Probability and GR Determination

We now describe how to determine the inside probabilities over the node sets N_i in the parse forest. We perform a depth-first search, filling GR specifications from word-nodes to the root node and returning the head and inside probability for each node in the extant parse forest, as described in §4.1.3. We determine the inside probability for each node n_c and node set N_i. That is, using Equation 4.9 and Equation 4.10 to calculate the inside probabilities $e(n_c)$ and $e(N_i)$, respectively.

We fill the GR specification for the node n_c using each head for the daughter node sets. That is, we use the head for $DN_l \in D(n_c)$ to fill the GR slot if the number l is specified in the unfilled GR. We restrict the set of parse forests considered to those in which each node $n_c \in N_i$ specifies the same lexical head. For each node n_c we determine a tuple $\{head(n_c), e(n_c)\}$ that represents the head and inside probability of the node. For each node set N_i, the tuple returned is $\{head(n_i), e(N_i)\}$, where n_i is the representative node for the set N_i. We defined previously that the head of a word node to be the word itself. Furthermore, we defined the inside probability of a word node to be the probability of the shift action which created the word node. Therefore, it is trivial to determine the head and inside probability for each node in the parse forest.

Example Sentence

For our example sentence, Table 4.1 illustrates the inside probability and filled GR specifications for each node in the parse forest shown in Figure 2.14. This table builds on Table 2.9 which illustrates the shift/reduce action probability and the corresponding instantiated (though

unfilled) GR specifications.[4.2] The filled GR specifications contain GRs that can be seen in the n-best GR sets and weighted GR output formats shown in Figure 2.16. For n_4 (`V1/v_np_pp`), we illustrate a pair of probabilities. The first is the inside probability of the node $e(n_4)$ and the second is for the node set $e(N_4)$. All three nodes of the set N_4 specify the same head `see+ed_VVD`. Thus, we pass up the pair $\{$`see+ed_VVD`$, -6.6284\}$ for this node set.

Outside Probability and Weighted GR Determination

Once we determine the inside probabilities and fill each GR specification for each node in the parse forest, we traverse the parse forest top-down to determine outside probabilities as discussed in §4.1.3. Table 4.2 illustrates the outside probability for each node (and node set) in the continued example. We sum over each possible outside probability (for each subanalysis in which the node is a daughter node) i.e. for each node n_c (row) of the table. This summation is shown for each mother node n_j, for the node in Equation 4.11. We consider the outside probability $f(n_c), \forall n_c \in N_i$ as the outside probability of the node set N_i, as shown in this equation. For example, we show a pair of probabilities for several columns of n_4, one for the node itself and one for N_4. The outside probability of N_4 is considered the outside probability for each node in the set, that is, n_4, n_{17} and n_{22}.

The IO probability of each node is the IO probability for the corresponding GR output by the node. This probability corresponds to (a portion of) the sum of all derivation probabilities that include the GR, that is, the non-normalised probability of the GR. In the case where the same GR occurs in different nodes of the parse forest, we sum over the IO for each of these nodes. That is, we apply the summation in Equation 4.13. To normalise the GR and determine $freq_q(g_d)$ as shown in this equation (the final weight of the GR), we divide by the inside probability of the root node. In our example, we show the normalised IO probability for each node (instance of a GR) then sum over these normalised probabilities instead. For example, the GR `(det man_NN1 the_AT)` results for nodes 6 and 23, summing over their (normalised IO) probabilities we weight this GR with value 1.

In practice, we store a hash table indexed against each GR. As we determine outside probabilities for each node, we (incrementally) store the IO probability in the hash table against each GR index. Thus after traversing the parse forest, the hash table contains a list of unique GRs and corresponding non-normalised weights. The non-normalised set of GRs for the example sentence is shown in Figure 2.16. In order to normalise this set we utilise the inside probability of the root node, that is, the probability of all derivations P_q. The final (normalised) set of weighted GRs for this example is also shown in Figure 2.16, where $P_q = -7.8438$.

4.2.3 Problem: Multiple Lexical Heads

The IOA$_{LR}$(1) solution applies only if a single lexical head results for each node, though this is often not the case. We extend the example parse forest discussed previously to illustrate how multiple heads may occur for each (tree) node in the parse forest. In related work, discussed in the following section, the parse forest is altered so that a single lexical head is guaranteed to result for each node. This allows these approaches to apply a similar solution as that described previously to determine weighted GR output.

Consider the set of nodes for the mother category `V1` in Figure 2.14, that is, nodes 4, 17 and 22. Each node specifies the head `see+ed_VVD`. However, if we alter the GR specification for

[4.2] Note that some values in this table, and others following for this example, differ slightly from the exact values that are calculated in this example in practice, due to rounding errors.

Node	Word/Rule	Prob	Inside	GR SPECIFICATION	
				head	GR(s)
1	T/txt-sc1/–	0.0	-7.8438	see+ed_VVD	
2	S/np_vp	-0.5391	-7.8438	see+ed_VVD	(ncsubj see+ed_VVD I_PPIS1 _)
3	I_PPIS1	-0.6763	-0.6763	I_PPIS1	
4	V1/v_np_pp	-0.8728	[-7.02172,-6.6284]	see+ed_VVD	(dobj see+ed_VVD man_NN1)(iobj see+ed_VVD in_II)
5	see+ed_VVD	-0.00002	-.00002	see+ed_VVD	
6	NP/det_n	-1.1568	-3.0052	man_NN1	(det man_NN1 the_AT)
7	the_AT	-0.0004	-.0004	the_AT	
8	N1/n	-1.848	-1.848	man_NN1	
9	man_NN1	0.0	0.0	man_NN1	
10	PP/p1	0.0	-3.1437	in_II	
11	P1/p_np	-0.6565	-3.1437	in_II	(dobj in_II park_NN1)
12	in_II	-0.0134	-0.0134	in_II	
13	NP/det_n	-0.1663	-2.4738	park_NN1	(det park_NN1 the_AT)
14	the_AT	-0.0005	0.0005	the_AT	
15	N1/n	-2.307	-2.307	park_NN1	
16	park_NN1	0.0	0.0	park_NN1	
17	V1/vp_pp	0.0	-8.66952	see+ed_VVD	(ncmod _ see+ed_VVD in_II)
18	V1/v_np	-2.5335	-5.53872	see+ed_VVD	(dobj see+ed_VVD man_NN1)
19	PP/p1	0.0	-3.1308	in_II	
20	P1/p_np	-0.6565	-3.1308	in_II	(dobj in_II park_NN1)
21	in_II	-0.0005	-0.0005	in_II	
22	V1/v_np	-1.1534	-6.86012	see+ed_VVD	(dobj see+ed_VVD man_NN1)
23	NP/det_n	-0.1663	-5.7067	man_NN1	(det man_NN1 the_AT)
24	N1/n1_pp1	-0.5165	-5.54	man_NN1	(ncmod _ man_NN1 in_II)
25	P1/p_np	-0.6565	-3.1755	in_II	(dobj in_II park_NN1)
26	in_II	-0.0452	-0.0452	in_II	

Table 4.1: Inside (log base 10) probabilities and filled GR specifications for parse forest nodes shown in Figure 2.14 and Table 2.9. The shift or reduce probability for the node is shown in the third column and the inside probability for the node is shown in the fourth column. The remaining two columns illustrate the filled GR specification for the node.

Node	Word/Rule	Inside	outside	IO	norm IO
1	T/txt-sc1/–	-7.8438	0.0	-7.8438	1.0
2	S/np_vp	-7.8438	0.0	-7.8438	1.0
3	I_PPIS1	-0.6763	-7.1675	-7.8438	1.0
4	V1/v_np_pp	[-7.02172,-6.6284]	-1.2154	[-8.23712,-7.8438]	[0.40428,1.0]
5	see+ed_VVD	-.00002	$\sum(-8.2371, -9.8849, -8.0755) = -7.8438$	-7.8438	1.0
6	NP/det_n	-3.0052	$\sum(-5.23192, -6.87972) = -5.2223$	-8.2275	0.41333
7	the_AT	-.0004	$\sum(-8.2271, -8.07512) = -7.84347$	-7.8438	1.0
8	N1/n	-1.848	$\sum(-6.3795, -6.22752) = -5.9959$	-7.84386	1.0
9	man_NN1	0.0	-7.84386	-7.84386	1.0
10	PP/p1	-3.1437	-5.0934	-8.2371	0.40428
11	P1/p_np	-3.1437	-5.0934	-8.2371	0.40428
12	in_II	-0.0134	-8.2237	-8.2371	0.40428
13	NP/det_n	-2.4738	$\sum(-5.7633, -7.41112, -5.60172) = -5.3700$	-7.8438	1.0
14	the_AT	0.0005	-7.8433	-7.8438	1.0
15	N1/n	-2.307	-5.5368	-7.8438	1.0
16	park_NN1	0.0	-7.8438	-7.8438	1.0
17	V1/vp_pp	-8.66952	-1.2154	-9.88492	0.009097
18	V1/v_np	-5.53872	-4.3462	-9.88492	0.009097
19	PP/p1	-3.1308	-6.75412	-9.88492	0.009097
20	P1/p_np	-3.1308	-6.75412	-9.88492	0.009097
21	in_II	-0.0005	-9.88442	-9.88492	0.009097
22	V1/v_np	-6.86012	-1.2154	-8.07552	0.58652
23	NP/det_n	-5.7067	-2.36882	-8.07552	0.58652
24	N1/n1_pp1	-5.54	-2.53552	-8.07552	0.58652
25	P1/p_np	-3.1755	-4.90002	-8.07552	0.58652
26	in_II	-0.0452	-8.03032	-8.07552	0.58652

Table 4.2: Outside and (non-)normalised IO probabilities for parse forest nodes shown in Figure 2.14 and Table 4.1.

V1/vp_pp to <2, (ncmod _ 1 2)>, then the second daughter is now considered the head. In this case, both V1/v_np_pp and V1/v_np specify the head see+ed_VVD, while V1/vp_pp specifies the head in_II. Thus for the mother node S/np_vp, we can fill the GR specification <2, (ncsubj 2 1)> as before, creating the filled GR specification: <see+ed_VVD, (ncsubj see+ed_VVD I_PPIS1)>. We may also fill the GR specification using the second daughter head combination:
<in_II, (ncsubj in_II I_PPIS1)>. It is clear in this example that node 2 (S/np_vp) appears in all possible derivations for the sentence. However, the two possible filled GR specifications clearly do not appear in every possible derivation. Consequently, we are no longer able to utilise the IO of the node within the frequency calculations for the filled GR specifications of the node. The new set of n-best GRs and weighted GRs is shown in Figure 4.4.

We describe the solution to this problem in §4.3. That is, we describe modifications to the $\text{IOA}_{LR}(1)$ described in this section. This algorithm is called 'EWG' as it is capable of *e*xtracting *w*eighted *G*Rs from *any* parse forest produced by the extant parser and is not limited to those in which a single lexical head results per node. EWG is based on the simple observation that if a set of nodes N_i specify more than one possible head, then the inside probability of each head forms part of the probability $e(N_i)$. We re-define the summation over nodes in N_i in Equation 4.10, so that we condition the summation on the node's head, where the function H returns the head of a node:

$$e(N_i) = \sum_h \sum_{n_c \in N_i, H(n_c)=h} e(n_c)$$

That is, we can determine the inside probability for each head h for N_i $e(N_i, h)$ using the inside probabilities for each node in the set where the previous equation can be split into two parts:

$$\begin{aligned} e(N_i, h) &= \sum_{n_c \in N_i, H(n_c)=h} e(n_c) \qquad (4.14) \\ e(N_i) &= \sum_h e(N_i, h) \end{aligned}$$

In the event that multiple heads occur for a node set, using the single probability $e(N_i)$ to represent the probability of each head over-estimates the probability $e(N_i, h)$ for a head, *unless* a single head only results for the node set. That is, the previous solution applies only if every node (set) has a single head.

4.2.4 Problem: Multiple Parse Forests

As described in §2.6.3, the grammar specifies a number of possible root categories, so a number of root node structures can result. Each of these structures define a set of derivations, where each derivation spans the whole sentence with the specified top category. In practice, we consider the parse forest as the set of such root node data structures.

In the following section, we describe EWG over a single root node, though the extension to multiple root nodes is trivial. To determine the inside probability of the sentence P_q, applied to normalise the weights of the weighted GR output, we sum over the inside probability of each root node produced. We perform the outside probability calculation by assigning the outside probability of *each* root node to be 1.

N-BEST GRS:

Parse probability: -8.075
(det man_NN1 the_AT)
(det park_NN1 the_AT)
(dobj in_II park_NN1)
(dobj see+ed_VVD man_NN1)
(ncsubj see+ed_VVD I_PPIS1 _)
(ncmod _ man_NN1 in_II)

Parse probability: -8.237
(det man_NN1 the_AT)
(det park_NN1 the_AT)
(dobj in_II park_NN1)
(dobj see+ed_VVD man_NN1)
(ncsubj see+ed_VVD I_PPIS1 _)
(iobj see+ed_VVD in_II)

Parse probability: -9.884
(det man_NN1 the_AT)
(det park_NN1 the_AT)
(dobj in_II park_NN1)
(dobj see+ed_VVD man_NN1)
(ncsubj in_II I_PPIS1 _)
(ncmod _ see+ed_VVD in_II)

(NON-NORMALISED) WEIGHTED GRS:
-7.843 (det man_NN1 the_AT)
-7.843 (det park_NN1 the_AT)
-7.843 (dobj in_II park_NN1)
-7.843 (dobj see+ed_VVD man_NN1)
-7.847 (ncsubj see+ed_VVD I_PPIS1 _)
-8.075 (ncmod _ man_NN1 in_II)
-8.237 (iobj see+ed_VVD in_II)
-9.884 (ncmod _ see+ed_VVD in_II)
-9.884 (ncsubj in_II I_PPIS1 _)

(NORMALISED) WEIGHTED GRS:
1.0 (det man_NN1 the_AT)
1.0 (det park_NN1 the_AT)
1.0 (dobj in_II park_NN1)
1.0 (dobj see+ed_VVD man_NN1)
0.990904 (ncsubj see+ed_VVD I_PPIS1 _)
0.5866 (ncmod _ man_NN1 in_II)
9.0960e-3 (ncmod _ see+ed_VVD in_II)
0.404265 (iobj see+ed_VVD in_II)
9.0960e-3 (ncsubj in_II I_PPIS1 _)

Total probability (sum of all parse probabilities): -7.843

Figure 4.4: The n-best GRs, and non-normalised/normalised weighted GRs for the three parses for the sentence *I saw the man in the park* using an altered GR specification in the rule `V1/vp_pp`. The corresponding parse forest and original GR set are shown in Figures 2.14 and 2.16, respectively.

4.3 The EWG Algorithm

In the previous section we illustrated the application of the $\text{IOA}_{LR}(1)$ over the extant parse forest to extract weighted GRs. As long as a single lexical head results for each node in the parse forest, the corresponding IO probability of the node may be used as the IO probability of the GR output by the node. However, in practice, more than one lexical head may result for each node in the parse forest. We describe the modifications made to our previous approach in subsequent sections, and discuss the relationship of our novel EWG algorithm to previous work in §4.3.3.

As described previously, three processing stages are required to determine weighted GRs over the parse forest. Several changes are required to the first stage of processing in which we

fill GR specifications and calculate inside probabilities. We discuss these changes in §4.3.1, and describe the data structure created during this stage. This structure is traversed during the next stage of processing, described in §4.3.2, to determine outside probabilities and non-normalised weighted GRs.

4.3.1 Inside Probability Calculation and GR Instantiation

Inside Data Structure

EWG allows multiple heads and their corresponding probabilities to propagate upwards in the parse forest. We return a set of data structures for each node set N_i, where each corresponds to a possible head for the node set. For now, we consider this data structure ($S^h_{N_i}$) to hold simply the head and corresponding inside probability. Thus, as before we pass up a tuple that consists of the node's head and inside probability. Although now more than one tuple is possible.

Multiple Heads in a Node Set

We first consider the case where each node has a single filled GR specification, and a filled GR specification in a packed node defines a different lexical head to that of the representative node. In this case, we allow *multiple heads* to be passed up by the node set. Hence, the summation over nodes in N_i in Equation 4.10 needs to be *conditioned* on the possible heads of a node, where $e(N_i,h)$ is the inside probability of each head h for node set N. That is we apply Equation 4.14:

$$e(N_i,h) = \sum_{n_c \in N_i, H(n_c)=h} e(n_c) \tag{4.15}$$

We create a data structure $S^h_{N_i} =< h, e(N_i,h) >$ for each head h for the node set N_i. For example, continuing the altered example in §4.2.3, we determine such a data structure for each head `see+ed_VVD` and `in_II`, returning for N_4 the list:

$$\{S^{\texttt{see+ed_VVD}}_{N_4} =< \texttt{see+ed_VVD}, -6.6324 >, S^{\texttt{in_II}}_{N_4} =< \texttt{in_II}, -8.66952 >\}$$

Here, the inside probability of the head `see+ed_VVD` is calculated using the inside probability of n_4 and n_{22}. That is, using the nodes whose GR specifications have the head `see+ed_VVD`. For the head `in_II`, the inside probability is that of the single node n_{17}.

Multiple Filled GR Specifications

We now need to consider multiple heads passed up by each daughter node, resulting in *multiple filled GR specifications* for a single node. We create one filled GR specification for each possible combination of daughters' heads. As described previously, for the mother node `S/np_vp`, we fill the GR specification <2, (ncsubj 2 1)> with each of the heads for N_4. This results in the filled GR specifications: <see+ed_VVD, (ncsubj see+ed_VVD I_PPIS1)> and <in_II, (ncsubj in_II I_PPIS1)>.

GR Specification Inside Probability

Each possible head for daughter node sets $DN_l \in D(n_c)$ has an associated inside probability, $e(DN_l,h)$, and we determine the probability of each filled GR specification, $s_m \in G(n_c)$ for a node n_c, using each daughter's head probability.

That is, we consider Equation 4.9 to apply to the lexical head of each daughter h_k chosen to fill the GR specification (in the set of possible lexical heads for daughter node set l: $H(DN_l)$):

$$e(n_c,s_m) = P(a[c]) \prod_{h_k \in H(DN_l), h_k \in s_m, DN_l \in D(n_c)} e(DN_l,h_k) \tag{4.16}$$

Returning to our previous example, we now calculate the inside probabilities of the GR specifications: <see+ed_VVD, (ncsubj see+ed_VVD I_PPIS1)> and <in_II, (ncsubj in_II I_PPIS1)>. These probabilities are equal to the reduce probability of the node n_2 (-0.5391) multiplied by (i) the inside probability of head `I_PPIS1` (-0.6763), and (ii) the inside probabilities of the heads `see+ed_VVD` (-6.6324) and `in_II` (-8.66952), respectively. That is, for the first filled GR specification, the inside probability is $\sum(-0.5391-0.6763-6.6324)=-7.8478$, while for the second it is $\sum(-0.5391-0.6763-8.66952)=-9.88492$.[4.3]

The same word can appear as a head for more than one daughter of a node. This occurs if competing analyses have daughters with different word spans and, therefore, particular words can be considered in the span of either daughter. As the grammar permits both pre- and post-modifiers, it is possible for words in the 'overlapping' span to be passed up as heads for both daughters. Therefore, semantic heads are not combined unless they are different words.

Head Inside Probability: Grouping GR Specifications

As a node can have multiple filled GR specifications $G(n_c)$, and packed nodes can also contain multiple filled GR specifications, we alter Equation 4.15 to:

$$e(N_i,h)=\sum_{n_c\in N_i}\sum_{s_m\in G(n_c),H(s_m)=h} e(n_c,s_m) \tag{4.17}$$

Here, $e(n_c,s_m)$ (the inside probability of filled GR specification s_m for node n_c) is determined using Equation 4.16. Hence, (a) calculation of inside probabilities takes into account multiple semantic heads, and (b) GR specifications are filled using every possible combination of daughters' heads.

Node Data Structures

For each node set N_i, which includes the representative node n_i and any packed nodes n_p, we propagate up the set of data structures $\{S^h_{N_i}\}$ for each possible head h for the node set. At word nodes, we simply return the word and the shift score of the node as the semantic head and inside probability, respectively.

$\{S^h_{N_i}\}$ also stores the corresponding set of GR specifications with the head h. That is, we store $G(N_i,h)=\{s_m\},\forall s_m\in G(n_c),n_c\in N_i,H(s_m)=h$. Furthermore, for each filled GR specification s_m which results for node n_c in the set N_i, we store the node's action probability $P(a[c])$ and inside probability of the GR specification $e(n_c,s_m)$.

$$S^h_{N_i}=(h,e(N_i,h),G(N_i,h)=\{(s_m:<h,\{GR\}>,P(a[c]),e(n_c,s_m),\{S^{h_k}_{DN_l}\}\}))$$

Note that we store the set of $S^{h_k}_{DN_l}$ structures for each daughter's head h_k used to fill a GR specification.

For example, for the node set N_4 in the continued example, we store the filled GR specifications <see+ed_VVD, (ncsubj see+ed_VVD I_PPIS1)> and <in_II, (ncsubj in_II I_PPIS1)> for each head `see+ed_VVD` and `in_II`, respectively. Thus for N_4 we return the set: $\{S^{\texttt{see+ed_NN1}}_{N_4},S^{\texttt{in_NN1}}_{N_4}\}$, where these data structures are shown in Figure 4.5. Similarly, we return the set for N_2: $\{S^{\texttt{see+ed_NN1}}_{N_2},S^{\texttt{in_NN1}}_{N_2}\}$, which are also shown in Figure 4.6.

[4.3] As previously described, summation of log probabilities is equivalent to the multiplication of these probabilities.

$$
\begin{aligned}
S_{N_4}^{\texttt{see+ed_NN1}} &= <\texttt{see+ed_VVD}, -6.6324, G(N_4, \texttt{see+ed_VVD})> \\
G(N_4, \texttt{see+ed_VVD}) &= \{(<\texttt{see+ed_VVD}, \\
&\quad \{\texttt{(dobj see+ed_VVD man_NN1),(iobj see+ed_VVD in_II)}\}>, \\
&\quad P(a[4]) = -0.8728, -7.02172, \{S_{N_5}^{\texttt{see+ed_VVD}}, S_{N_6}^{\texttt{man_NN1}}, S_{N_{10}}^{\texttt{in_II}}\}) \\
&\quad (<\texttt{see+ed_VVD}, \{\texttt{(dobj see+ed_VVD man_NN1)}\}>, \\
&\quad P(a[22]) = -1.1534, -6.86012, \{S_{N_5}^{\texttt{see+ed_VVD}}, S_{N_{23}}^{\texttt{man_NN1}}\})\} \\
S_{N_4}^{\texttt{in_II}} &= <\texttt{in_II}, -8.66952, G(N_4, \texttt{in_NN1})> \\
G(N_4, \texttt{in_II}) &= \{(<\texttt{in_II}, \{\texttt{(ncmod _ see+ed_VVD in_II)}\}>, \\
&\quad P(a[17]) = 0.0, -8.66952, \{S_{N_{18}}^{\texttt{see+ed_VVD}}, S_{N_{19}}^{\texttt{in_II}}\})\}
\end{aligned}
$$

Figure 4.5: Example EWG data structures for N_4.

$$
\begin{aligned}
S_{N_2}^{\texttt{see+ed_NN1}} &= <\texttt{see+ed_VVD}, -7.8478, G(N_2, \texttt{see+ed_VVD})> \\
G(N_2, \texttt{see+ed_VVD}) &= \{(<\texttt{see+ed_VVD}, \{\texttt{(ncsubj see+ed_VVD I_PPIS1)}\}>, \\
&\quad P(a[2]) = -0.5391, -7.8478, \{S_{N_3}^{\texttt{I_PPIS1}}, S_{N_4}^{\texttt{see+ed_VVD}}\})\} \\
S_{N_2}^{\texttt{in_NN1}} &= <\texttt{in_NN1}, -9.88492, G(N_2, \texttt{in_NN1})> \\
G(N_2, \texttt{in_NN1}) &= \{(<\texttt{in_NN1}, \{\texttt{(ncsubj in_NN1 I_PPIS1)}\}>, \\
&\quad P(a[2]) = -0.5391, -9.88492, \{S_{N_3}^{\texttt{I_PPIS1}}, S_{N_4}^{\texttt{in_NN1}}\})\}
\end{aligned}
$$

Figure 4.6: Example EWG data structures for N_2.

Overview

Each node set N_i, of node n_i with packed nodes n_p, is processed in full as follows:

- Process each of the node's packed nodes n_p (as described for n_i following) to determine the packed node's list of filled GR specifications and corresponding inside probabilities.

- Process the node n_i, with daughters $D(n_i)$:

  - Instantiate n_i's GR specifications based on features of daughters $D(n_i)$.
  - Process each daughter node set in $DN_l \in D(n_i)$ to determine a list of possible heads and corresponding inside probabilities for each. That is, we determine the set of $S_{DN_l}^{h_k}$ structures for each daughter node set.
  - Fill the GR specification of n_i with each possible combination of daughters' heads, creating the set $G(n_i)$.
  - Calculate the inside probability of each filled GR specification $e(n_i, s_m)$, that is, for each $s_m \in G(n_i)$ using Equation 4.16.

- Combine the alternative filled GR specifications of n_i and of each of the packed nodes n_p, to determine the list of unique semantic heads h and corresponding inside probabilities $e(N_i, h)$ using Equation 4.17.

- Create $S^h_{N_i}$ for each unique h and return the set of such structures in a list: $\{S^h_{N_i}\}$

This results in the data structure over the parse forest, that is, over the set of parse forests with root node N_r: $S_\tau = \{S^h_{N_r}\}$. This structure is traversed during the next stage of processing, so that the parse forest itself is only traversed once.

4.3.2 Outside Probability Calculation

GR Specification Outside Probability

After the inside probabilities are computed (bottom-up), the resulting data structure S_τ is traversed to compute outside probabilities. This data structure is already split into alternative heads for each node set N_i ($S^h_{N_i}$). Therefore, it is trivial to traverse this structure to determine outside probabilities. The outside probability of $S^h_{N_i}$ is equal to the outside probability of each filled GR specification data structure stored in $S^h_{N_i}$. That is, $f(n_c, s_m) = f(N_i, h)$ for each possible s_m filled for node $n_c \in N_i$, where the head of the GR specification s_m is h.

Daughter Node Outside Probability

We then calculate the outside probability of each daughter node $DN_l \in D(n_c)$, whose head h_k was used to fill the GR specification s_m resulting for node n_c. We consider the outside probability of the GR specification to be that of the mother node in Equation 4.11. Further, we utilise the inside probability of each daughter node DN_p for the head h_r, a fellow daughter, which also fills the GR specification s_m. Note that the set of probabilities $e(DN_p, h_r)$ for each k and r is already stored within the data structure with s_m. The test we define for this product ($h_r \in s_m$) is not required if we apply this equation within the data structures created during the previous processing stage.

$$f(DN_l, h_k) = \sum_{n_c} f(n_c, s_m) P(a[c]) \prod_{DN_p \in D(n_c), DN_p \neq DN_l, h_r \in s_m} e(DN_p, h_r) \tag{4.18}$$

For example, for N_4 we calculate the outside probability of the head `see+ed_VVD`, where $f(n_2, <$ `see+ed_VVD`$, \{$`(ncsubj see+ed_VVD I_PPIS1)`$\} >) = 0$, $P(a[2]) = -0.5391$ and $e(N_3,$ `I_PPIS1`$) = -0.6763$[4.4] as:

$$f(N_4, \texttt{see+ed_NN1}) = 0 + -0.5391 + -0.6763 = -1.2154$$

Considering that $f(n_4, s_m) = f(N_4,$ `see+ed_NN1`$)$ for GR s_m with head `see+ed_NN1`, we determine the non-normalised weight of the GRs for n_4 $\{$`(dobj see+ed_VVD man_NN1)`, `(iobj see+ed_VVD in_II)`$\}$ as -1.2154 + -7.02172 = -8.23712. Normalising each GR by -7.8438, we find, for example, the final weight of `(iobj see+ed_VVD in_II)` is 0.40428.[4.5]

[4.4] The $\log_{10}$ probability of 1 is 0. Again, we use summation to perform multiplication for log probabilities.

[4.5] Again, these figures differ slightly from weight shown in Figure 4.4, due to rounding errors.

Traversing the New Data Structure

Therefore, once we create the new data structure, outside probabilities for each node are determined over this structure in the regular fashion, rather than over the parse forest. We simply equate the outside probability of each head to be that of every corresponding filled GR specification data structure.

Overview

In practice, we apply a breadth first search (FIFO queue) over S_τ, to minimise multiple processing of shared data structures. We initialise this queue to contain each root node with an outside probability of 1. We perform a POP operation to determine the next paired list from the queue. This pair consists of the data structure and corresponding outside probability: $\{S^h_{N_i}, f(N_i,h)\}$. We process each pair as follows:

- Process each of the GR specifications in $(S^h_{N_i})$. For each filled GR specification s_m, where $s_m \in G(N_i,h)$ was created for node $n_c \in N_i$:
  - Let $f(n_c,s_m) = f(N_i,h)$ and calculate the IO probability of s_m using:

$$IO(n_c,s_m) = e(n_c,s_m) f(n_c,s_m) \tag{4.19}$$

  - Add $IO(n_c,s_m)$ to the (non-normalised) probability for the GR in the filled GR specification s_m.
  - Process the data structure for each daughter head h_k chosen to fill s_m from the daughter node set $DN_l \in D(n_c)$. That is, process each of the daughter data structures (that each specify a single head) used to fill the slots in the GR specification s_m. For each $S^{h_k}_{DN_l}$ stored for s_m:
    * Calculate the outside probability $f(DN_l,h_k)$ of the head h_k from daughter node set DN_l using the reduce probability of the node n_c which created s_m: $P(a[c])$, which is also stored in the data structure. That is, we use Equation 4.18 without the summation over n_c; the set of mother nodes for which DN_l is a daughter.
    * Queue the data structure $S^{h_k}_{DN_l}$ and corresponding outside probability $f(DN_l,h_k)$.

We calculate the outside probabilities (top-down) and, when we find filled GR specifications, we incrementally store the non-normalised (IO) weight of each GR in a hash table, as described previously. Each increment represents the situation where the node is a daughter of another in the parse forest. That is, performs the summation over mother nodes n_c in Equation 4.18. Thus, if an outside probability is determined for a data structure already queued, then the probability is appended to the list of outside probabilities for the queued item and Prior to calculating Equation 4.19 we perform a summation over each outside probability, where $f(n_c,s_m) = \sum f(N_i,h)$.

Traversing the Parse Forest

We could instead store the inside probabilities and the set of filled GR specifications within each node of the parse forest, rather than create the S data structures. In this case, we calculate outside probabilities *conditioned* on which lexical head (from each daughter) is used to fill each GR specification. Here, we pass down the outside probability $f(N_i,h)$ to each node $n_c \in N_i$ and then to each of the GR specifications stored in n_c if and only if the head of the GR specification

is h. This test is not required when processing over S_τ as GR specifications are already grouped by head value. Equation 4.18 is already shown with the condition that we include the inside probability $e(DN_p, h_r)$ for daughter p if and only if h_r filled a slot in s_m.

4.3.3 Related Work

In the previous sections we illustrated that a dynamic programming approach over the parse forest can be used to calculate weighted GR output *only* if a single head is determined for each node in the parse forest. Further we illustrated how to extend this application to successfully associate *multiple* heads, and corresponding inside and outside probabilities, with each node in the parse forest using our novel EWG algorithm.

The approach we take is similar to that in Schmid & Rooth (2001), where 'expected governors' (similar to our filled GR specifications) are determined for each node. However they ensure that competing nodes (i.e. each node in a node set) in the parse forest have the *same head*. Initially, they create a packed parse forest and during a second pass the parse forest nodes are split if multiple heads occur. A single iteration of the IOA is applied over this split PCFG parse forest data structure, as described for LR parse forests using the IOA$_{LR}$(1) in §4.2.2. However, as they utilise a PCFG grammar, rules specify how to determine dependency relations given tuples consisting of the NT categories of mother and daughters of CFG rules. In contrast, within the extant parser such rules are encoded in finer grained unification-based rules within the grammar itself. Moreover, these are optionally instantiated on the features of the rules' daughters.

Clark & Curran (2004b) apply the dynamic programming approach of Miyao & Tsujii (2002) to determine weighted dependency relations (DR) within their CCG parser. They alter their packing algorithm so that nodes in the packed chart effectively have the same semantic head. That is, their definition for feature structure equivalence is extended to include the head of each node as well.

These previous approaches determined dependency relations and corresponding weights directly from the parse forest, using dynamic programming approaches similar to the IOA$_{LR}$(1) we describe. That is, they ensure that a single head is specified for each node, which results in a less compact parse forest. As a result, they require additional processing overheads to create the parse forest, and moreover, to determine inside and outside probabilities from this less compact parse forest.

In order to apply the EWG algorithm, we also modify the packing algorithm to be based on feature structure equality rather than subsumption. This results in a less compact parse forest, though only to the extent originally found in the parse forests of Schmid & Rooth (2001) and Clark & Curran (2004b). If we extend the definition of node equivalence to include the node's head (of a filled GR specification), or split the nodes in a postprocessing stage as the previous approaches have, this increases the size of the parse forest further. Therefore, we consider our approach as an efficient alternative to those in previous work, given that many existing parsers already utilise equality-based packing.

4.4 EWG Performance

This section presents experimental results showing (a) improved efficiency and (b) increased upper bounds of precision and recall achieved using EWG. In the following section, §4.5, we show (c) increased accuracy achieved by a parse selection algorithm that would otherwise be too inefficient to consider. We utilise the same data (DepBank) and machine hardware described

in 3.5.1.

4.4.1 Comparing Packing Schemes

Figures 4.7 and 4.8 compare the efficiency of EWG to the EQ-PACKING and SUB-PACKING methods in terms of CPU time and memory, respectively.[4.6] Note that EWG applies equality-based packing to ensure only derivations licensed by the grammar are considered.

As the maximum number of (n-best) derivations increases, EQ-PACKING requires more time and memory than SUB-PACKING. However, if we compare these systems with an n-best value of 1, the difference in time and memory is negligible. This suggests that it is the unpacking stage which is responsible for the decreased throughput. For EWG we are forced to use equality-based packing. Although these results suggest that this condition does not affect the algorithm's throughput.

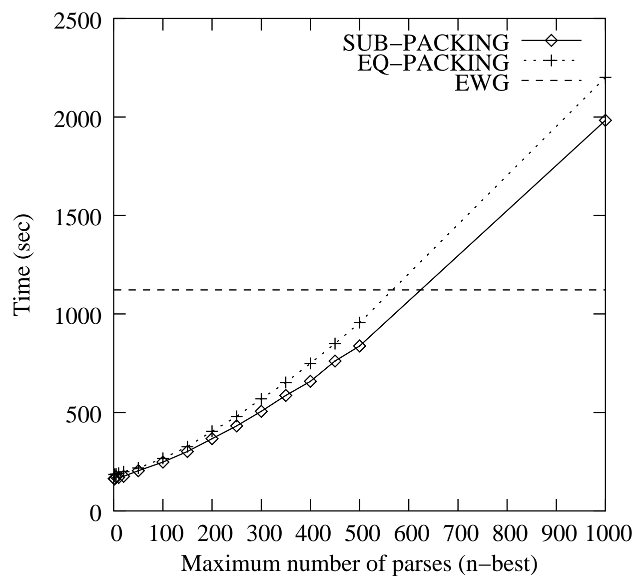

Figure 4.7: Comparison of total CPU time required by the different versions of the parsing system for calculation of weighted GRs over the n-best derivations.

4.4.2 Efficiency of EWG

Both figures illustrate that the time and memory required by EWG are static, because the algorithm considers all derivations represented in the parse forest regardless of the value of n-best specified. Therefore, the 'cross-over points' are of particular interest: at which n-best value is EWG's efficiency the same as that of the current system's? This value is approximately 580 and 100 for time and memory, respectively (comparing EWG to EQ-PACKING). That is, the current system takes the same amount of time to calculate weighted GRs over approximately 580 n-best parses as the EWG algorithm takes to calculate weighted GRs directly from the parse forest (over all parses). Similarly, the memory requirements are approximately equal for the current system utilising an n-best list of size 100. Consequently, as the size of the n-best list unpacked (for the existing system's calculation of weighted GRs) increases over these cross-over

[4.6] CPU time and memory usage are as reported using the `time` function in Allegro Common Lisp 7.0 and do not include system start-up overheads or the time required for garbage collection.

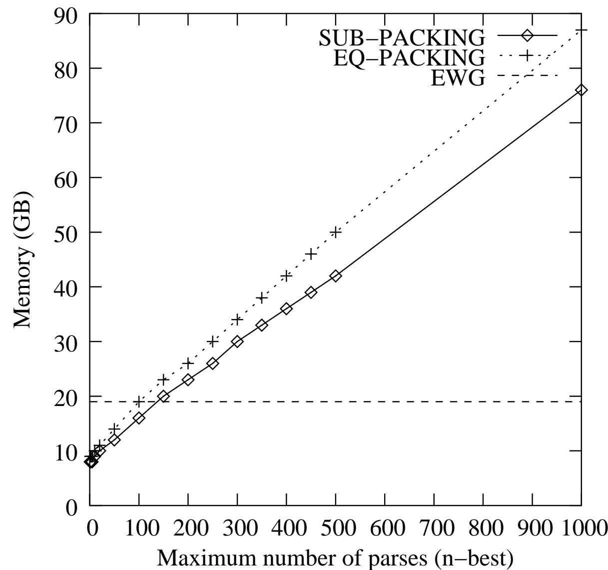

Figure 4.8: Comparison of total memory required by the different versions of the system for calculation of weighted GRs over the n-best derivations.

points, the EWG algorithm becomes more efficient in comparison to the existing system (for the given dataset).

Given that there are on average around 9K derivations per sentence for DepBank, these results indicate a substantial improvement in both efficiency and accuracy for weighted GR calculation. However, the median number of derivations per sentence is around 50, suggesting that large derivation numbers for a small subset of the test suite are skewing the arithmetic mean. Therefore, the complexity of this subset significantly decreases throughput, and EWG improves efficiency for these sentences more so than for others.

4.4.3 Data Analysis

The general relationship between sentence length and number of derivations suggests that the EWG is more beneficial for longer sentences. Figure 4.9 shows the distribution of derivation ambiguity over sentence length. The figure illustrates that the number of derivations can not be reliably predicted from sentence length, though the general relationship holds. Considering the cross-over points for time and memory, the number of sentences with more than 580 and 100 derivations were 216 and 276, respectively. These points are shown using dotted lines in the figure. Thus, the EWG outperforms the current algorithm for around half of the sentences in the data set. The relative gain achieved overall for EWG reflects that a subset of sentences significantly decreases throughput. Hence, the EWG is expected to be more efficient than the current method for determining weighted GRs if longer sentences are present in the data set and if the size of the n-best list is set to a value greater than the cross-over point(s). Table 1.1 illustrates the average and median number of parses for a number of different data sets.[4.7] Consequently, EWG is expected to outperform the current method over all corpora except the simple GDT.

[4.7] We employ a version of the tsg15 grammar in these experiments that is less ambiguous than the one released with version 2 of RASP (applied to determine the statistics in Table 1.1). As a result, EWG is expected to provide even further performance gains over the current method for determining weighted GRs.

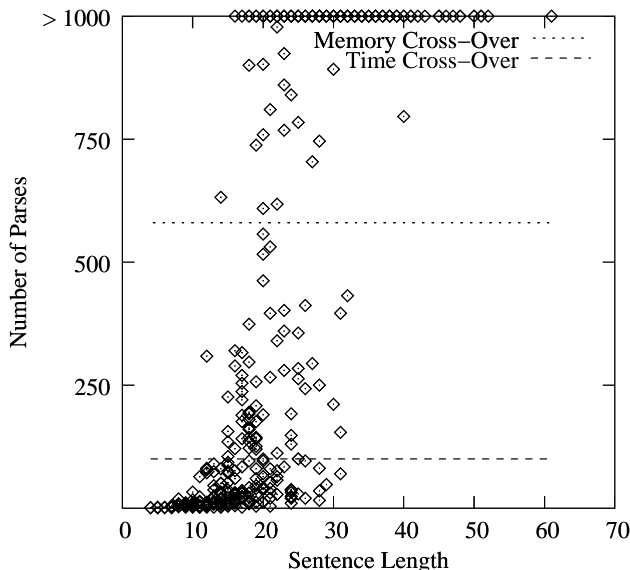

Figure 4.9: Scatter graph of parse ambiguity to sentence length (one point per sentence). The cross-over points are illustrated for time and memory. The maximum number of derivations shown is 1K, points plotted at 1K correspond to sentences with equal to or greater than 1K derivations.

4.4.4 Accuracy of EWG

Upper bounds on precision and recall are determined using thresholds over the weighted GRs of 1 and 0, respectively.[4.8] Upper bounds of precision and recall provided by EWG are 79.57 and 82.02, respectively. This results in an F_1 upper bound of 81.22%. However, considering the top 100 derivations only, we achieve upper bounds on precision and recall of 78.77% and 81.18% respectively, resulting in an F_1 upper bound of 79.96%. Therefore, using EWG, we achieve a relative increase of 6.29% for the F_1 upper bound on the task. Similarly, Carroll & Briscoe (2002) demonstrate (on an earlier, different test suite) that increasing the number of derivations (n-best) from 100 to 1K increases precision of weighted GR sets from 89.59% to 90.24%, a relative error reduction (RER) of 6.8%. Therefore, EWG achieves a substantial improvement in both efficiency and accuracy for weighted GR calculation. The approach provides an increase in the upper bounds of precision and recall, that is, it provides an increased F_1 upper bound on the task.

4.5 Application to Parse Selection

The previous section illustrated the increased level of efficiency achieved by EWG compared to the extant method for calculating weighted GRs. This section describes a parse selection algorithm using EWG that would otherwise be too inefficient to apply.

We previously described the work of Clark & Curran (2004b), whereby weighted dependency relations (DRs) are determined directly from a packed chart. They also describe a parse selection algorithm which maximises the expected recall of dependencies. Their algorithm se-

[4.8] In these experiments we use a threshold of $1-\varepsilon$ (with $\varepsilon = 0.0001$) instead of a threshold of 1.0, to reduce the influence of very low ranked derivations. Note that these upper bounds are at least as high as those achieved by selecting the best possible single n-best GR set.

lects the DR set with the highest average DR score based on the weights from the weighted DRs.

We apply this parse selection algorithm in two ways. We (a) rerank the n-best GR sets based on the average weight of GRs and select the highest ranking set, or (b) apply a simple variant of the Viterbi algorithm to select the GR set with the highest average weighted score over the data structure built during EWG. The latter approach, based on the parse selection algorithm in Clark & Curran (2004b), takes into account *all possible derivations* and effectively reranks these derivations using weights output by EWG. These approaches are referred to as *RERANK* (over the top 1K derivations) and *BEST-AVG*, respectively. Table 4.3 illustrates the performance of the extant parsing model and of these approaches over DepBank.

	Micro-average			Macro-average		
Model	Prec	Rec	F_1	Prec	Rec	F_1
Extant	71.24	71.24	71.24	58.42	59.62	59.01
RERANK	71.72	70.94	71.33	59.47	60.22	59.84
BEST-AVG	71.59	71.49	71.54	59.53	60.63	60.08

Table 4.3: Performance of the two parse selection algorithms.

The GR set corresponding to the system's top parse achieves an F_1 of 71.24%. By applying BEST-AVG and RERANK parse selection, we achieve a relative error reduction (compared to the upper bound of 81.22%) of 3.01% and 0.90%, respectively. Therefore, BEST-AVG achieves higher accuracy and is more efficient than RERANK. This suggests that BEST-AVG is able to select derivations that are ranked in a position lower than 1K. Thus, as expected, it is advantageous to consider all derivations during reranking tasks. It is also worth noting that these parse selection schemes are able to output a *consistent* set of GRs, unlike the high precision GR output defined by Carroll & Briscoe (2002).

4.6 Discussion

We described a dynamic programming approach based on the IOA for producing weighted GR output directly from the parse forest of the extant unification-based LR parser. Our approach is novel in that we allow any node in the parse forest to have *multiple heads*. This removes the additional processing overheads, introduced by either splitting nodes in the parse forest (and duplicating the associated data structures) or from building a less compact parse forest, as described in previous approaches. In an evaluation on a standard test suite (DepBank), the approach achieves substantial improvements in accuracy and parser throughput over the extant implementation. EWG is available for use within the second release of the RASP parser, as an alternative method to calculate the weighted GR output format.

We intend to extend this work to develop more sophisticated parse selection schemes based on weighted GR output. Reranking the n-best GR sets results in a consistent but not necessarily a complete set of GRs. Given the (increased) upper bound on precision for the high precision GR output, we hope to boost the corresponding recall measure by determining a consistent and complete set of GRs directly from the weighted GR set. We continue this discussion in Chapter 6.

Chapter 5

Confidence-based Training

We discussed statistical approaches to parsing in §1.1.3, where we defined training a parser as a *supervised* learning task. The current training method for RASP, described in §5.3, utilises such a supervised training framework. It employs around 5K of fully-annotated and system compatible derivations from a subset of Susanne to derive the action counts, thus corresponding probabilities, of the LR model. However, such treebanks are limited in their use and coverage, and furthermore, are expensive to develop and maintain. These limitations, discussed in detail in §5.1, have prompted the development of *unsupervised* and *semisupervised* statistical training approaches that illustrate promising results.

In this chapter we focus on semisupervised training approaches, reviewed in §5.2, and describe novel *confidence-based* training methods, and their relationship to previous work, in §5.4. We contrast the performance of the extant parser trained over Susanne, with that of the same parser trained over unannotated or unlabelled partially-bracketed sentences from this treebank and from the WSJ. That is, we contrast the performance of the current, fully supervised, training method to that of the unsupervised or semisupervised confidence-based methods in §5.5.

We also compare the performance of these methods against the IOA_{LR}, the IOA extended to apply over our extant LR parser as described in §4.1.3, and further, to apply over unlabelled partially-bracketed data following Pereira & Schabes (1992). We consider the IOA_{LR} a variant of Expectation-Maximisation (EM), following Prescher (2001). Hence, in the experimentation described herein, we refer to the IOA_{LR} training methods applied as EM.

The semisupervised variants of the confidence-based methods we describe outperform both EM (when both are constrained over the same data set) and the current fully supervised training method, achieving statistically significant improvements over the extant parser. Though we would expect a supervised method to outperform a semisupervised one over the *same* data set, these results suggest that a semisupervised method can outperform a supervised one given sufficient training data. As semisupervised data is simpler to extract from existing, though incompatible, corpora and also simpler to develop, these methods are preferable to fully supervised methods and aid in adapting the parser to new domains.

Although our training methods are considered semisupervised, they utilise partially-annotated data automatically extracted from existing corpora. As a result, they require no manual effort on behalf of the grammar writer. These methods have been adopted by the extant parser, as they provide significant increases in accuracy, and furthermore, aid in grammar development. Much of the work we describe in this chapter appears in Watson *et al.* (2007).

5.1 Motivation

Currently, many statistical parsers require extensive and detailed treebanks, as many of their lexical and structural parameters are estimated in a fully-supervised fashion from treebank derivations. Collins (1999) is a detailed exposition of one such ongoing line of research, which utilises the standard training sections of the WSJ. However, there are disadvantages to this approach which we discuss in this section.

Manual Effort

Firstly, treebanks are expensive to create manually. Further, given a manually written grammar, the grammar writer must also *maintain* these treebanks. For example, §1.3.1 described the annotated (supervised) training corpus of the extant parser that incorporates a manually written feature-based unification grammar (see §2.6.1). Grammar updates require equivalent (manual) updates to the annotation in this training corpus, where interactive manual disambiguation takes an average of ten minutes per sentence.

Usability

Secondly, the richer the annotation required, the harder it is to adapt the treebank to train parsers which make different assumptions about the structure of syntactic analyses. For example, Hockenmaier (2003) trains a CCG statistical parser on the WSJ, but first maps the treebank to CCG derivations semi-automatically.

Coverage

Thirdly, many (lexical) parameter estimates do not generalise well between domains as discussed in §1.1.3. For instance, Gildea (2001) reports that WSJ derived bilexical parameters in Collins (1999) Model 1 parser contribute about 1% to parse selection accuracy when test data is in the same domain. In contrast, they yield no improvement for test data selected from the Brown Corpus (of which Susanne is a subset). However, training over both corpora slightly increases performance when testing on either corpus.

Therefore, the use of in-domain training data improves parsing accuracy. In-domain data can be included with out-of-domain data in a single training corpus. Alternatively, the parser may be *adapted* to the new domain by retraining the parser over a separate corpus, using the existing parser as the *initial model*. Here, even small levels of in-domain data are shown to improve parser accuracy. Note that we can frame the problem of *parser tuning* or *domain adaptation* as an unsupervised or (semi)supervised task.

For example, Tadayoshi *et al.* (2005) adapt a statistical parser trained on the WSJ to the biomedical domain by retraining on the Genia Corpus, augmented with manually corrected derivations in the same format. They are able to tune their parser with around 5K of in-domain annotated sentences (a small set compared to 40K of WSJ sentences that train the parser initially).

Discussion

In order to make statistical parsing more viable for a range of applications, we need to make more effective and flexible use of existing training data and minimise the cost of annotation for new data created to tune a system to a new domain. Therefore, the focus of this work is to develop *semisupervised* training approaches that utilise partially-annotated data extracted from existing, though incompatible, treebanks. Further, these methods can apply over unsupervised in-domain data, to adapt the parser to a new domain if required.

5.2 Research Background

Training methods can be defined based on the level of supervised annotation. That is the level of manual effort required, to provide training data prior to or during the training process. If we utilise fully annotated training data we consider the approach to be *supervised*. The limitations of these approaches have prompted the development of unsupervised and semisupervised methods.

Although unsupervised methods have proven relatively unsuccessful, semisupervised methods have illustrated promising results. We provide examples of several relatively successful unsupervised training methods in §5.2.1, and review the popular semisupervised approaches in §5.2.2.

5.2.1 Unsupervised Training

Training approaches that require no (manual) annotation are considered *unsupervised*. These methods are generally based on the IOA, which we described in the previous chapter, and have been largely unsuccessful to date. Although an advantage of such approaches is that raw data in any domain is readily available.

In recent years, inferring the grammar and statistical model from unlabelled data has provided some encouraging results. For example, Klein & Manning (2002) report promising results for unsupervised grammar induction over the ATIS Corpus (Marcus *et al.*, 1993) and section 10 of the WSJ. The unlabelled F_1 of 71% achieved for the WSJ section is only around 10% lower than the performance of a supervised PCFG trained over the same section. Klein & Manning (2004) extend this work, and combine the constituency based model with a dependency model, which achieves a further increase in performance to 77.6%.

Similarly, Bod (2006) illustrates that an unsupervised 'DOP' parser trained using EM related to the IOA ('UML-DOP') outperforms a supervised PCFG when both are trained over the WSJ. However the underlying models differ, and we still expect (semi)supervised training to outperform unsupervised training given the *same* model and data set.

5.2.2 Semisupervised Training

Given limited (in-domain) training data or manual effort available, two general types of semisupervised training methods are pursued in research, either increase the *quantity* and/or *quality* of training data while minimising the corresponding required manual effort. Approaches can consist of a mixture of (i) using an existing, though incompatible, treebank's annotation, (ii) *active learning*, and (iii) *bootstrapping* methods. We describe each of these approaches (and their variants), and research which focuses on these methods, in this section. These approaches utilise either partially annotated data or a mixture of both supervised and unsupervised data.

Mapping Treebanks

Given an existing corpus, with different representational assumptions, we can semi-automatically map the corpus to a representation compatible with the parser's grammar. For example, Hockenmaier (2003) trains a CCG statistical parser on the WSJ, but first maps the treebank to CCG derivations semi-automatically, creating the *CCGBank*. Similarly, Miyao & Tsujii (2005) semi-automatically map the WSJ to a head-driven phrase structure grammar (HPSG) treebank. The resulting treebanks are fully annotated, facilitating supervised training. However, the semi-automatic mapping requires less manual effort compared to the creation of these corpora from

scratch, so we consider the underlying method to be semisupervised. These methods are more labour intensive than those that follow.

Partial Treebank Annotation

Alternatively we can utilise only a subset of the annotation that is potentially more compatible with the parser's grammar. Pereira & Schabes (1992) adapt the IOA to apply over semisupervised data, that is, over unlabelled bracketing, that they extract from the WSJ. They constrain the training data (derivations) they consider within the IOA to those consistent with the constituent boundaries defined by the bracketing. One advantage of this approach is that, although less information is derived from the treebank, it generalises better to parsers which make different representational assumptions. Furthermore it is easier, as Pereira and Schabes did, to map unlabelled bracketing to a format more consistent with the target grammar. Another is that the cost of annotation with unlabelled brackets is lower than that of developing a representationally richer treebank.

More recently, Riezler *et al.* (2002) illustrate a method to train a maximum entropy parsing model over semi-supervised data. They utilise all derivations in the model consistent with the labelled bracketing of the WSJ. They first extract a partial annotation from the WSJ treebank that consists of the PoS tags and labelled brackets that are relevant for determining their GRs. In order to utilise this annotation, they map from WSJ PoS tags to the terminals of their grammar. Parsing over these PoS tags results in a number of possible derivations, as the extracted annotation is not sufficient to disambiguate the finer grained LFG derivations. They modify the standard log-linear objective function so that they include all such consistent derivations (normalising by the set of all derivations). In contrast to the approach of Pereira & Schabes (1992), this approach incorporates all derivations licensed by the grammar during normalisation, and not only those consistent with the partial annotation.

Clark & Curran (2004b), following Riezler *et al.* (2002), train their maximum entropy model for their CCG parser using all derivations consistent with the gold standard derivations in the CCGBank created by Hockenmaier (2003). However, in Clark & Curran (2006), they utilise partially annotated data only, where the training data consists only of supertags. They extend the work of Clark & Curran (2004b), and redefine the consistency of a derivation in terms of the corresponding dependency relations. They consider the set of dependencies that occur in at least k% of all derivations (licensed by the set of supertags in the training data) to be a gold standard dependency set. This set of dependencies is analogous to the set of high precision weighted GRs produced by RASP created using a threshold of $k/100$. This approach achieves impressive performance; only 1.2% worse on section 23 of the WSJ than the model trained on full annotations.

Hwa (1999) applies the IOA to (partially) labelled data following Pereira & Schabes (1992). However these experiments focus on the level of bracketing and constitute type (labels of the brackets), over the ATIS and WSJ corpora, from which the grammar may be reliably learnt. This experimentation illustrates that higher level constituents, that constitute only a small proportion of the bracketing, are the most informative. In fact, to train the underlying grammar, the model requires only 25% (randomly selected) of the bracketing to achieve over 90% accuracy for ATIS. In contrast, if the model trains over all brackets, it achieves around 93% accuracy. Similarly, on the WSJ, the model achieves within 3% of the fully supervised performance when it trains over 25% of (randomly selected) brackets. Results are slightly lower if Hwa infers the grammar as well as the statistical model.

Active Learning

Active learning (AL) is a framework for *actively* selecting samples that should be manually annotated. The primary goal of AL is to reduce the number of annotated training sentences needed to achieve the same level of parsing accuracy. A popular method for AL is *selective sampling* (Cohn *et al.*, 1994), whereby the AL system selects the next set of unannotated samples (from a large unannotated corpus) to be annotated. For natural language parsing, we wish to select the set of sentences which is most informative. Approaches favour either *certainty-* or *committee*-based methods.

For certainty based methods, an initial (optionally supervised) parser selects the set of unannotated sentences with the lowest confidence, e.g. derivation probability. For committee-based approaches, a set of parsers is used to annotate the unannotated sentences. The set of sentences that led to the highest number of parser disagreements is selected.

A number of different certainty- and committee-based approaches are explored in the literature, which we do not cover here. Instead, we refer readers to Osborne & Baldridge (2004), who provide references to research in this area and compare the performance of a number of these approaches. They illustrate that, in general, certainty-based sampling slightly outperforms committee-based sampling. Furthermore, they show that both methods clearly outperform selection methods that apply random sampling, sentence length and overall structural ambiguity (number of derivations).

Bootstrapping

Bootstrapping significantly increases the quantity of training data, where additional training data is *automatically* annotated using an initial statistical model. These methods combine a (small) *seed* set of manually labelled data, with a (large) set of unlabelled *bootstrap* data which is automatically labelled. If the seed and bootstrap data are from different domains, then we consider this a *domain adaptation* method. If the seed data is also unannotated, then the approach is unsupervised.

These methods generally require manual effort to create the initial seed data only, unless an existing treebank is utilised to train the initial model. In either case, such methods are considered semisupervised due to the annotation provided in the seed data set. There are a number of variants to the general bootstrapping framework. For example, the way in which the data sets are combined may differ.

Self-training

In *self-training* approaches, we train an initial parsing model over the seed data. This parser then labels the bootstrap data set, that is, the parser labels its own training data. Charniak (1997) is the first to apply this method, though he did not use this terminology. In general, the top-ranked derivation for each sentence in the bootstrap data is added to the training data as though the parser's annotation is correct. That is, counts are simply merged between the seed and labelled bootstrap data sets.

Initial research in these methods illustrates that either modest performance gains, or significant decreases occur, in parsing and PoS tagging models. Charniak (1997) reports minor gains if he retrains a parser over 30 million words of unsupervised WSJ text. That is, he uses an initial 'basic' model to annotate these sentences. Similarly, Clark *et al.* (2003) illustrate that self-training with PoS taggers results in unimpressive performance, regardless of the size of the initial seed data set.

This framework incorporates any number of iterations. If we use more than one iteration, then we repeatedly update the initial model by labelling the entire bootstrap data or only the next portion of this data. Furthermore, we can set the size of each portion depending on the number of iterations to perform. Steedman *et al.* (2003b) illustrate modest improvement and a steady decline in performance when performing self-training for each of two different parsers, respectively. Each parser is initialised with the first 500 sentences from the standard WSJ training sections. Using self-training iterations, each iteration incorporates 30 sentences of which 20 of those with the highest top-derivation probabilities are included. Therefore, this self-training method utilises some selective sampling within each iteration.

Recently, successful instances of self-training have been published in the literature. Bacchiani *et al.* (2006) report the first successful self-training experiments. Out-of-domain labelled seed data (from the Brown Corpus) is used to train the initial model which is then applied over the in-domain, though unannotated, WSJ bootstrap data (representing an unsupervised domain adaptation task). A single training iteration is performed over the entire bootstrap data set. The set of (up to 20) candidate derivations for each sentence, each weighted by their normalised derivation probability, is included during training. This results in weighting features of the model (corresponding to each derivation) using weights analogous to the expected frequency counts for the IOA, as described previously in §4.1.2. They simply combine the counts from each corpus, though set the contribution of the in-domain bootstrap data to be 5 times that of the out-of-domain seed data (this weight was optimised on supervised data). The performance of the self-trained parser increases from 75.70% to 80.55% f-score over section 23 of the WSJ, as the number of WSJ sentences in the bootstrap data increases from 0 to 200K. Furthermore, this work shows that during supervised domain adaptation, simply merging counts from each corpus outperforms model interpolation. That is, it outperforms interpolation between different parsing models where each model is trained over a separate corpus. Consequently, they utilised count-merging during the unsupervised domain adaptation.

Similarly, McClosky *et al.* (2006) report successful self-training experiments over a discriminative reranking parser. Here, an initial generative parser outputs the 50-best list, which is reranked by a discriminative maximum entropy model. Following Bacchiani *et al.* (2006), counts from each domain are simply merged where in-domain (WSJ) data is weighted 5 times higher. However, they utilise only the top-ranked parse output by either the generative or discriminative parsing system during self-training and consider these top-derivations to be 'correct'. The (baseline) reranking system achieves 91.3% f-score when trained and tested over the standard WSJ sections. They illustrate that self-training using the generative model alone, i.e. the generative parser's top-parse, results in a decline in performance. However, incorporating the top-ranked derivations output by the full reranking model improves parser performance to 92.1%, currently the best reported PARSEVAL accuracy for the WSJ.

Co-training

Co-training is another variant of bootstrapping in which parsers annotate *each other's* bootstrap training data iteratively. Generally, co-training outperforms self-training methods, even when one parser initially achieves much lower accuracy than another. This method requires that the annotation from both parsers are (somewhat) compatible.

Co-training has been used in a number of natural language processing applications including word-sense disambiguation, web-page classification and named-entity identification. Sarkar (2001) performs the first application of co-training to parsing, using two components of a single

lexicalised tree adjoining grammar (LTAG) parser; the supertagger and the parser. A PoS tag dictionary is constructed over the labeled and unlabeled training data which is used by both the supertagger and the parser. Initial models for the supertagger and parser are created by training over a small initial data set of around 10K (seed) sentences from sections 2-6 of the WSJ. Sarkar co-trains over around 30K unannotated (bootstrap) sentences extracted from sections 7-21, and evaluates the models over the de facto WSJ section 23. The co-trained parsing model achieves 80.02% and 79.64% precision and recall, respectively, while the initial parsing model (trained over the seed data) achieves only 72.23% and 69.13%, respectively. Therefore, co-training is able to significantly improve the initial model's performance. These results are not as significant as those in previous applications, though parsing is arguably harder than those tasks which involve a smaller and simpler set of labels and relatively small parameter spaces.

Steedman *et al.* (2003b) extend the work of Sarkar (2001) by co-training using *separate* parsers: the (same) LTAG parser and Model 2 of Collins (1999). They effectively combine the co-training framework with the AL selective sampling method, selecting 20 sentences of 30 during each iteration as previously described. Their results show that co-training outperforms self-training significantly, and is most beneficial when the labelled seed data sets are smaller (500 compared to 1K sentences). Though they note that the performance of co-training from 500 seed sentences and an additional 2K of co-labelled bootstrap sentences, never achieves the accuracy of the initial model trained over 1K seed sentences (76.9% compared to 78.6%).

Furthermore, performance achieved by co-training over the Brown Corpus is around 2% lower (76.8% vs. 79.0%) than that achieved over the WSJ. This performance drop is in agreement to that noted by Gildea (2001). However if they seed the data with an additional 100 sentences from the WSJ, then the resulting co-trained performance increases to 78.2%, close to that achieved by seeding the model with in-domain data only. Therefore, these experiments help to illustrate that limited in-domain data can help parser accuracy. Steedman *et al.* (2003a) continue from Steedman *et al.* (2003b), and investigate alternative scoring functions within the selective sampling methods in each co-training iteration.

Corrected Co-training

The training frameworks discussed thus far can be combined to produce a wide-range of training architectures. For instance, Pierce & Cardie (2001) propose a semisupervised variant of co-training, which Steedman *et al.* (2003a) apply, termed *corrected co-training*. This model attempts to combine the strengths of co-training and AL. During each co-training iteration, selected samples are manually verified and corrected if required. Steedman *et al.* (2003a) compare the performance of co-training and corrected co-training, where they apply the same sampling method in both cases. That is selective sampling determines the next set of bootstrap sentences to annotate, and these sentences are either annotated manually or automatically by the (other) parser. As expected, corrected co-training significantly reduces the number of bootstrap training sentences required to achieve the same level of parser accuracy. Though corrected co-training requires additional manual effort.

5.3 Extant Parser Training and Resources

In this section we review the available training corpora in §5.3.1, and extant parser training in §5.3.2, which we described in full within previous chapters. We define *derivation consistency* over a fully supervised corpus as applied by the extant parser, and over a semisupervised (unlabelled partially-bracketed) corpus as described by Pereira & Schabes (1992). Finally, we review

the evaluation methods in §5.3.3, and following, describe the baseline system in this work and its performance in §5.3.4.

5.3.1 Corpora

Corpus Format

In §1.3.1, we described a treebank T which consists of a set of training instances. Each training instance t is a pair (s,M), where s is the automatically preprocessed text (tokenised and labelled with PoS tags (see §3.2.2) and M is either a fully annotated derivation, A, or an unlabelled bracketing U. We extend this definition to include unsupervised corpora in which M is null.

In this section we define the set of training corpora, and following, derivation consistency for each treebank type where either A or U is paired with each sentence in a corpus. The bracketing in U may be partial in the sense that it may be consistent with more than one derivation produced by a given parser. For unsupervised training, all derivations are considered 'consistent' given that no annotation is provided. Note that the sentence s which is parsed during training, consists entirely of automatically preprocessed text using the RASP pipeline in all corpora considered. That is, we do not utilise the PoS tags in either Susanne or the WSJ.

Training Corpora

We previously described fully annotated and bracketed corpora for Susanne and the WSJ in §1.3.1. We utilise these corpora in the experimentation described in this chapter, and also the combination of both bracketed Susanne and WSJ corpora. We use the sentence-delimited preprocessed text from Susanne during unsupervised training.

We refer to the fully annotated corpus created from a subset of Susanne, the extant training data, as B, as this represents the *baseline* training corpus. B consists of 4801 training instances in the format (s,A).

The bracketed corpora extracted from Susanne and the WSJ are referred to as S and W. These corpora consist of 7014 and 38,329 training instances, respectively, in the format (s,U). The concatenated file containing both Susanne and WSJ bracketed training instances is referred to as SW. We refer to the unsupervised variant of the S corpus as S_u. However, in practice, we process the S file and ignore the bracketing paired with each sentence s.

Annotated Derivation Consistency

Given $t = (s,A)$, there exists a single derivation in the parse forest that is compatible (correct). In this case, equality between the derivation tree and the treebank annotation A identifies the single correct (consistent) derivation.

Bracketed Derivation Consistency

Following Pereira & Schabes (1992), given $t = (s,U)$, a node's span in the parse forest is *valid* if it does not overlap with any span defined in U. A derivation is considered *consistent* if the span of every node in the derivation is valid in U. That is, if no crossing brackets are present in the derivation. Each valid node in the parse forest is considered consistent if one or more possible daughter node sets are also consistent and if it is a member of a consistent derivation.[5.1]

[5.1] When we walk the parse forest during the EM training methods we collect consistent node's actions and the corresponding IO probability. We walk the parse forest along valid paths, so that we only consider the set of nodes in the parse forest that appear in one or more consistent derivations.

For example, Figure 1.3 illustrates an example bracketed lemmatised training instance from Susanne. The (single) derivation output by the extant parser for this sentence is shown in Figure 5.1. The corresponding constituent boundaries: *(((his (petition)) (charge (((mental)) (cruelty)))))* are consistent with the unlabelled-bracketing extracted from Susanne shown in Figure 1.3: *((his petition) charge (mental cruelty)*.

```
(T/txt-sc1/-+
 (S/np_vp (NP/det_n1 his_APP$ (N1/n petition_NN1))
  (V1/v_n1 charge_VVN
   (N1/ap_n1/- (AP/a1 (A1/a mental_JJ))
    (N1/n cruelty_NN1))))
 (End-punct3/- ._.))
```

Figure 5.1: Example RASP output for a sentence from Susanne.

Given the set of consistent nodes in the parse forest, or set of consistent derivations within the n-best list, we extract the corresponding action histories and estimate action probabilities, as described in §2.6.2. We extract the corresponding action for each consistent node in the parse forest (see §4.1.3). Alternatively, for each node in a consistent derivation, where actions may occur in more than one derivation. In this way, partial bracketing is used to constrain the set of derivations (and thus, corresponding LR parse actions) considered in training to those that are compatible with this bracketing.

5.3.2 Extant Parser Training

In this section we briefly review theory regarding supervised training for parametric models, and that of the extant parser, described fully in previous chapters.

Training Parametric Models

As described previously, a supervised corpus (or *treebank*), T consists of a set of training instances where each training instance $t_i \in T$ is a pair (s,A). The parse forest Y is created by parsing over s, representing the set of derivations $y_{ij} \in Y$. A single derivation in the parse forest $y_c \in Y$ is equal to the derivation A, and therefore, is considered the 'correct' derivation. The set of features for each correct derivation y_c, is extracted for each training instance $t_i \in T$.

The features' frequencies are used to determine MLE for the model, often using relative frequency estimates over sets of competing features. For example, the set of rule rewrites for a given NT category of a PCFG. For an LR parser, the features are parsing actions and we normalise the frequency of these actions based on the total frequency across each set of competing actions in the LR table.

LR Parser Training

Estimating action probabilities in the LR table (see §2.5.2) consists of (a) recording an action history for the correct derivation y_c, for each training instance $t_i \in T$, (b) computing the frequency of each action over all action histories and (c) normalising these frequencies to determine probability distributions over conflicting actions.

For supervised training, the weight of each action in the first step is considered to be 1. That is, we consider step (b) a weighted frequency sum over actions where each action witnessed has a weight of 1. Therefore, to determine the frequency of action a_d in the LR table, where

the function $HIST$ returns the set of LR parsing actions used to create the given derivation, we apply the following equation where $\delta(x,y)$ returns 1 if x equals y and 0 otherwise:

$$freq(a_d) = \sum_{t_i \in T} \sum_{a \in HIST(y_c)} \delta(a, a_d) \quad (5.1)$$

Extant Parser Training

The extant parser (our model trained using the extant supervised method described in §2.6.2), is considered the baseline system in this work. We determine action counts by training on B. That is, we apply Equation 5.1 over the action histories for each correct derivation in this supervised corpus. We normalise over these action counts using the normalisation method defined by Inui *et al.* (1997), as discussed in §2.5.2. In addition, we apply Laplace estimation within this normalisation method in the extant parser, to ensure all actions in the table are assigned a non-zero probability (the I_L function). In view of these definitions, we consider the baseline system in this work to be the parsing system $I_L(B)$.

5.3.3 Evaluation

We employ DepBank (see §1.3.1) as test data in subsequent experiments to compare parser performance. Further we utilise the Wilcoxon test for statistical significance (see §1.3.2) and provide z-values probabilities to compare parsing systems, predominantly against the baseline system.

5.3.4 Baseline

The micro-averaged F_1 score for the baseline system over DepBank is 75.61%, which (over similar sets of relational dependencies) is broadly comparable to recent evaluation results published by Kaplan *et al.* (2004) with their state-of-the-art parsing system (Briscoe & Carroll, 2006).

5.4 Confidence-based Training Approaches

In this section we describe *confidence-based* training. The general framework of this approach, and its relationship to previous approaches, is defined in §5.4.1. Following, in §5.4.2, we describe a number of confidence measures, which we apply in the experimentation discussed in §5.5. Finally we define a self-training variant of the framework in §5.4.3 which is also applied in experimentation to follow.

5.4.1 Framework

The general confidence-based training framework is described in this section, and is closely related to the self-training frameworks discussed. We describe in subsequent sections, the initial parsing model which is trained over seed data, and then applied to annotate the bootstrap data. The differentiating factor of our framework is the incorporation of alternative *confidence* measures which depend on the initial parsing model used.

We also extend the self-training framework to include varying levels of annotation in both the seed and bootstrap data sets. As a result, we perform unsupervised or semisupervised training, where the level and type of annotation may vary. Given any type of annotation, we require only a definition of *derivation consistency*, as we define our models over the set of consistent derivations for each sentence in the bootstrap data set.

Note that the confidence-based framework can also be extended to include co-training techniques, and therefore, applied to a wide range of parsing models and training methods. We describe a single-pass over the entire bootstrap data only. However, we could extend the framework so that the initial model is repeatedly updated. This extension is analogous to the iterative self-training methods described.

Initial Parsing Model

The first stage in our framework simply involves training the statistical parser over the initial seed data. While in self-training the seed data is generally a small supervised corpus, any initial parsing model can be considered. Therefore, we can train the initial model over an arbitrarily large seed data set (including null) using semisupervised or unsupervised training techniques.

Annotating Derivations

The self-training methods previously described utilise an initial parsing model to *annotate* remaining unlabelled data. That is, they parse the unlabelled data with the initial parser and take the top-ranked parse produced for each sentence as additional training data. Therefore, these methods effectively consider the annotation of this top ranked parse as correct. The final parsing model is created by training over both the seed corpus and the annotated bootstrap corpus in a fully supervised fashion. In contrast, our models consider *all* derivations produced by the initial model, and weight the contribution of the corresponding features of the derivation based on the initial model's *confidence* in each derivation.

If we apply the confidence training framework over unsupervised seed and bootstrap data, then this method is related to the work of Bod (2006), which was published after the work in this chapter was complete. Bod applies an unsupervised variant of 'self-training' with the 'DOP' parser. The parser is trained with simple frequency estimates over the top 100 derivations, 'U-DOP', which performs 2% worse than a supervised PCFG when both are trained over the WSJ. However, while all derivations are considered in these frequency estimates, each of the derivations are weighted *equally*. We include all derivations in the confidence-based methods, though we weight them according to the initial model's confidence in the derivations returned.

If the bootstrap data consists of semisupervised (that is, partially annotated) data, then we follow previous work (e.g. Pereira & Schabes 1992) and restrict the set of derivations we include to those that are consistent with this annotation. Effectively, we apply any partial annotation as a means to 'filter' out the derivations in the n-best list that are incompatible with this annotation. This increases the quality of the derivations considered during training. Thus, the higher the level of annotation in the bootstrap data, the better our methods are expected to perform.

Estimating Action Probabilities

The corresponding features of *all* derivations within the n-best list (optionally, only those consistent with annotation, if available) are utilised during training. However, we weight the features of the model based on the corresponding *derivation confidence* of the initial parsing model used to derive the n-best list. A variant of Equation 5.1 is applied over the set of consistent derivations, where we instead apply a *weighted* sum. The weight is determined using the function c over the derivations in which the feature appears. We determine the frequency of each action in the LR parse table using the weighted frequency of each occurrence of the action across all $t_i \in T$ in each consistent derivation $y_i \in Y$:

$$freq(a_d) = \sum_{t_i \in T} \sum_{y_i, a_d \in HIST(y_i)} c(y_i) \tag{5.2}$$

The frequency of each feature determined over the bootstrap data is combined directly with the frequency of each feature over the seed data i.e. using count-merging.

Relationship to the IOA (EM)

We described the extension of the unsupervised IOA to LR parsers in §4.1.3; the IOA_{LR}. The frequency Equation 4.1.3 can be considered in terms of the confidence measure function c, applied to nodes n_j of the parse forest as follows:

$$\begin{aligned} freq(a_d) &= \sum_{t_i \in T} \sum_{n_j, a[j]=a_d} c(n_j) \\ c(n_j) &= \frac{1}{P_{t_i}} e(n_j) f(n_j) \end{aligned} \tag{5.3}$$

Equation 5.3 is related to the confidence-based frequency Equation 5.2, where we sum over derivations rather than nodes of the parse forest. Recall that the normalised IO probability for a node represents the summation of probabilities for all derivations in which the node occurs, normalised by the sum of all derivation probabilities. Consequently, if we unpack *all* possible derivations represented in the parse forest, weighted by their normalised probabilities, then the resulting weighted frequency counts for each unique node in these derivations is equal to the corresponding parse forest node's normalised IO probability.

If we utilise a function c in Equation 5.2, that returns the derivation's normalised probability (as in Bacchiani *et al.* 2006) we effectively perform one iteration of the IOA_{LR}, though over the set of n-best derivations rather than the entire derivation space. Thus the confidence based methods require only a small proportion (equivalent to a single iteration) of the processing overhead required to train using IOA_{LR}.

Relationship to Previous Work

In summary, the framework defined is related to, though differentiated from, previous work described as follows:

- Self-training: while we utilise an initial model to annotate further training data, we consider unsupervised and semisupervised training for the initial parsing model. Moreover, we include the set of *all* derivations (consistent with the annotation, if available) of the bootstrap data rather than only the highest ranked derivation.

- Partially-annotated data: we constrain the set of derivations considered to those that are consistent with the partial annotation. However, we construct and rank this derivation set using an *initial parsing model* trained over a separate seed data set. Previous approaches that have constrained the set of derivations use only a uniform parsing model to construct and rank the derivations. That is, include counts from the (single) bootstrap data only to create the resulting parsing model.

- The work of Bacchiani *et al.* (2006), regarding unsupervised self-training, is closely related to our framework, though was published after the work in this chapter was complete. A feature's weight, i.e. the function c, is the corresponding derivations' normalised probability. This weighting effectively determines the frequency of features over the bootstrap data by applying one iteration of unsupervised IOA, considering the 20-best derivations only, instead of the entire parse forest.

In effect, we have combined the successful semisupervised training approaches that (i) constrain the set of derivations considered to those consistent with partial annotation (e.g. Clark & Curran 2006; Pereira & Schabes 1992; Riezler *et al.* 2002, with (ii) self-training approaches (e.g. Bacchiani *et al.* 2006; McClosky *et al.* 2006). Although we generalise the framework to include unsupervised training and apply confidence based weighting within the frequency counts for features.

5.4.2 Confidence Measures

During confidence-based training, we select more than one derivation, placing an appropriate weight on the corresponding action histories based on the initial model's confidence in the derivation. In this section we define three such confidence measures for consistent derivations (i.e. returned for the function c in Equation 5.2). In subsequent experimentation we contrast the performance of each against EM, over both unsupervised and semisupervised corpora.

We weight transitions corresponding to each derivation ranked r with probability p in the set of size n either using $\frac{1}{n}$, $\frac{1}{r}$ or p itself to weight counts. These methods all perform normalisation over the resulting action histories using the training function I_L (defined in §5.3.2) and are referred to as C_n, C_r and C_p, respectively. These functions take two arguments: an initial model and the bootstrap data to train over, respectively. As we improve the accuracy of the initial model, and decrease the size of the n-best list in response, the accuracy of the resulting parsing model is expected to increase across all of these measures. We discuss each measure, in turn, in the following sections.

Uniform Measure: C_n

C_n is a 'uniform' model which weights each action count only by the degree of derivation ambiguity and makes no explicit use of ranking information. In effect, the initial parser only acts to provide the n-best derivations, where the likelihood of a correct parse being in this set increases as the accuracy of the initial model increases. While we weight in a uniform manner, both the initial parsing model and the number of derivations considered (the size of the n-best list) affect the accuracy of the resulting trained parser. This method is similar to that of Riezler *et al.* (2002), where all consistent derivations are included in the log-linear objective function. However, we normalise using consistent derivations only.

Ranking Measure: C_r

C_r weights each action count using the corresponding derivation's rank. This measure is based on the intuition that features that consistently occur in highly ranked derivations are more likely to be correct, and hence, should be assigned a greater proportion of the probability mass.

Probability Measure: C_p

C_p weights each action count using the derivations' probability. This measure places the greatest level of trust in the initial model's statistical component. Given a 'perfect' initial parsing model, C_p is expected to outperform all other possible measures, as it assigns probability mass to correct syntactic subanalyses only. C_p is simpler than and different to one iteration of IOA_{LR}, as we use inside probabilities only, and furthermore, do not normalise based on the sum of all derivation probabilities.

In the case that we consider an n-best list of size 1, this method is considered similar to Viterbi training for HMMs, where the top-ranked path's probability is used in the weighted frequency sum for corresponding edges in the path.

5.4.3 Self-training

In experimentation to follow, we also perform a variant of self-training within the confidence training framework. That is, we assign a weight of 1 to each action corresponding to the top-ranked parse output by the initial model over the bootstrap training corpus. We refer to this training method as C_1.

This method is closely related to the self-training methods employed in the literature which we discussed previously, though we first 'filter' the set of n-best derivations. That is, we consider only those derivations that are consistent with the partial-annotation of the sentence.

5.5 Experimentation

As we utilise an initial model to annotate additional training data, our methods are closely related to self-training methods. However, in experimentation discussed in this section, we train entirely from either unannotated or unlabelled partially-bracketed data using the confidence training framework described in the previous section. Therefore, these methods are best described as unsupervised or semisupervised, respectively. We expect the extant parser trained over a fully supervised corpus to outperform one trained over the same corpus with less detailed annotation. However, both EM (IOA_{LR}) and confidence-based methods are trained over *larger* semisupervised (and unsupervised) corpora, providing greater potential for these methods to outperform the extant parser.

In §5.5.1 we contrast these models over semisupervised corpora consisting entirely of unlabelled partially-bracketed data. We utilise such data extracted from Susanne and the WSJ, as a major focus of this work is the flexible reuse of existing treebanks to train a wider variety of statistical parsing models. We train a different parsing model for each of the confidence measures (described in §5.4.2), and for the self-training method (described in §5.4.3). Here, the confidence-based training achieves statistically significant improvements in parser accuracy over both EM and the current supervised training method. In addition, these methods are more efficient than EM and require no manual annotation effort on behalf of the grammar writer.

Following, in §5.5.2, we compare the unsupervised variants of EM and the confidence based models over the unsupervised Susanne corpus S_u. Surprisingly both models perform only slightly worse than the extant fully supervised method, moreover, these differences are not statistically significant (if we select the best performing EM iteration).

5.5.1 Semisupervised Training

In this section, we compare the accuracy of the current parse ranking model trained from a fully-annotated portion of Susanne with one trained from unlabelled partially-bracketed training instances derived from this treebank and from the WSJ. We demonstrate that the confidence-based semisupervised techniques outperform EM when both are constrained by partial bracketing. Both methods based on partially-bracketed training data outperform the fully supervised technique, and both can, in principle, be applied to any statistical parser whose output is consistent with such partial-bracketing. We also explore tuning the model to a different domain and the effect of in-domain data in the semisupervised training processes.

Experimental Setup

We parsed all the bracketed training data using the baseline model to obtain up to 1K top-ranked derivations and found that a significant proportion of the sentences of the potential set available for training had only a single derivation consistent with their unlabelled bracketing. We refer to

this set of sentences as the *unambiguous training data* (γ) and refer to the remaining sentences (for which more than one consistent derivation was returned) as the *ambiguous training data* (α). The availability of significant quantities of unambiguous training data that can be found automatically suggests that we may be able to dispense with the costly reannotation step required to generate the fully supervised training corpus, B.

Table 5.1 illustrates the split of the corpora into mutually exclusive sets γ, α, 'no match' and 'timeout'. The latter two sets are not utilised during training and refer to sentences for which all derivations were inconsistent with the bracketing and those for which no derivations were found due to time and memory limitations (self-imposed) on the system, respectively.[5.2] As our grammar is different from that implicit in the WSJ there is a high proportion of sentences where no derivations were consistent with the unmodified PTB bracketing. However, a preliminary investigation of the 'no match' data did not yield any clear patterns of inconsistency that we could quickly ameliorate by simple modifications of the PTB bracketing. We leave for the future a more extensive investigation of these cases which, in principle, would allow us to make more use of this training data. An alternative approach that we have also explored is to utilise a similar confidence-based training approach with data partially-annotated for grammatical relations (Watson & Briscoe, 2007).

Corpus	$\mid\gamma\mid$	$\mid\alpha\mid$	No Match	Timeout
S	1097	4138	1322	191
W	6334	15152	15749	1094
SW	7409	19248	16946	1475

Table 5.1: Corpus split for S, W and SW.

Confidence-based Approaches

Models derived using the unambiguous training data, γ, as the seed data alone are relatively accurate, achieving indistinguishable performance to that of the baseline system given in-domain (either W or SW) training data. We utilise these models as initial models and train over the corresponding ambiguous data sets (considered the bootstrap data) for each corpus with each of the confidence-based models. We consider the top 1K derivations output by each of the initial models over the bootstrap data, and then remove derivations from this n-best list that are inconsistent with the corresponding unlabelled bracketing.

The initial models are novel, being the first to consider the use of such unambiguous data only. These models were based on the intuition that, as only a single derivation is consistent with the annotation, we can assume that the partial-annotation is sufficient to disambiguate the finer grained derivations of the extant parser. This data can be considered analogous to the seed data set used during the self-training methods in the literature, where a supervised (high confidence) data set is used to train the initial model.

Table 5.2 gives results for all confidence-based models. Results statistically significant compared to the baseline system are shown in bold print (increase) or italic print (decrease). These methods show promise, often yielding systems whose performance is significantly better than the baseline system. Method C_r achieved the best performance in this experiment and remained

[5.2] As there are time and memory restrictions during parsing, the SW results are not equal to the sum of those from S and W analysis.

consistently better across different corpora. Throughout the different approaches a domain effect can be seen, models utilising just S are worse, although the best performing models benefit from the use of both S and W as training data (i.e. SW). These results are consistent with those found by Gildea (2001).

The 'self-training' C_1 method performs the worst of all retrained models, resulting in a *decline* in performance over the initial model for both W and SW. The increase in performance for this model over S may reflect the benefit of using a balanced corpus during training, though may be due to the relatively small size of the seed data considered where there is less chance for bias in the model to be reflected in the reannotated data.

While many statistical parsers extract the grammar in parallel with the corresponding statistical parse selection model, our results demonstrate that existing treebanks can be utilised to train parsers that deploy grammars that make other representational assumptions. As a result, our methods can be applied by a range of parsers to minimise the manual effort required to train a parser or adapt to a new domain.

Model	Prec	Rec	F_1	$P(z)^{\ddagger}$
Baseline	77.05	74.22	75.61	-
$I_L(\gamma(S))$	76.02	73.40	*74.69*	0.0294
$C_1(I_L(\gamma(S)),\alpha(S))$	77.05	74.22	75.61	0.4960
$C_n(I_L(\gamma(S)),\alpha(S))$	77.51	74.80	76.13	0.0655
$C_r(I_L(\gamma(S)),\alpha(S))$	77.73	74.98	**76.33**	0.0154
$C_p(I_L(\gamma(S)),\alpha(S))$	76.45	73.91	75.16	0.2090
$I_L(\gamma(W))$	77.01	74.31	75.64	0.1038
$C_1(I_L(\gamma(W)),\alpha(W))$	76.90	74.23	75.55	0.2546
$C_n(I_L(\gamma(W)),\alpha(W))$	77.85	75.07	**76.43**	0.0017
$C_r(I_L(\gamma(W)),\alpha(W))$	77.88	75.04	**76.43**	0.0011
$C_p(I_L(\gamma(W)),\alpha(W))$	77.40	74.75	76.05	0.1335
$I_L(\gamma(SW))$	77.09	74.35	75.70	0.1003
$C_1(I_L(\gamma(SW)),\alpha(SW))$	76.86	74.21	75.51	0.2483
$C_n(I_L(\gamma(SW)),\alpha(SW))$	77.88	75.05	**76.44**	0.0048
$C_r(I_L(\gamma(SW)),\alpha(SW))$	78.01	75.13	**76.54**	0.0007
$C_p(I_L(\gamma(SW)),\alpha(SW))$	77.54	74.95	76.23	0.0618

Table 5.2: Performance of confidence-based training models on DepBank. ‡represents the statistical significance of the system against the baseline model.

Comparing EM and Confidence-based Approaches

As previously noted, we consider the IOA_{LR} a variant of EM following Prescher (2001). In order to further extend the IOA_{LR} to apply over semisupervised corpora, following Pereira & Schabes (1992), we simply have to extend Equation 5.3 (i.e. Equation 4.1.3) so that only consistent nodes are included in this summation. We employ the function τ over nodes, which returns 1 if a node is consistent (see §5.3.1) and 0 if not.

$$freq(a_d) = \sum_{t \in T} \sum_{n_j, a[j]=a_d} \tau(n_j) c(n_j)$$

We perform EM starting from two initial models; either a uniform probability model, $I_L()$, or from models derived from unambiguous training data, γ. We create the uniform model by distributing the probability mass equally between competing actions in the LR table. This model achieves 69.92%, which is fairly good given that no training data has been incorporated into the statistical model, indicating the expressive power of the underlying manually written grammar.

We denote the EM models using the function EM. This function accepts the same input as the confidence based functions, that is, an initial model and the bootstrap data to train over, respectively. We utilise the cross entropy estimate described in Pereira & Schabes (1992), where the cross entropy H over a given corpus C and grammar G is based on P_t; the total probability of all derivations for each sentence $t \in C$:

$$H(C,G) = -\frac{\sum_{t \in C} log(P_t)}{\sum_{t \in C} |t|}$$

Figure 5.2 shows the cross entropy decreasing monotonically from iteration 2 (as guaranteed by the EM method) for different corpora and initial models. Some models show an initial increase in cross-entropy from iteration 1 to iteration 2, because the models are initialised from a subset of the data which is used to perform maximisation. Cross-entropy increases, by definition, as we incorporate ambiguous data with more than one consistent derivation (i.e. increasing the ratio of $log(P_t)$ to $|t|$).

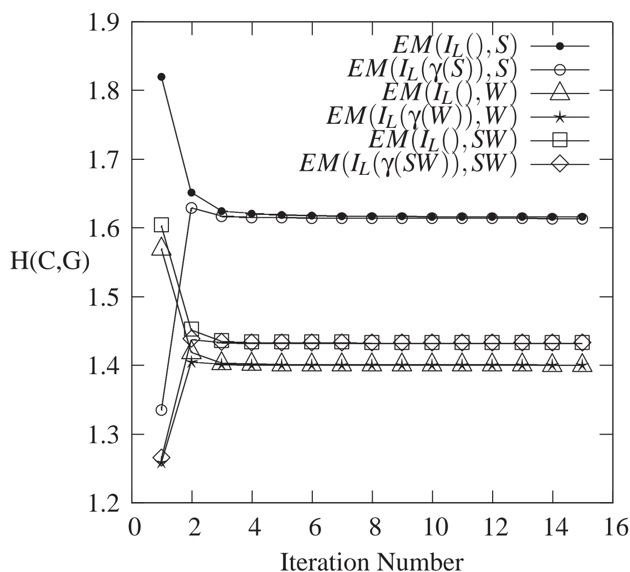

Figure 5.2: Cross entropy convergence for semisupervised EM over S, W and SW.

Performance over DepBank can be seen in Figures 5.3, 5.4, and 5.5 for each corpus S, W and SW, respectively. Comparing the accuracy of C_r and EM in each of Figures 5.3, 5.4, and 5.5, it is evident that C_r outperforms EM across all data sets, regardless of the initial model applied. In most cases, these results are statistically significant, even when we manually select the best model (iteration) for EM.

The graphs of EM performance illustrate the same 'classical' and 'initial' patterns observed by Elworthy (1994) (described in §3.1.3). When EM is initialised from a relatively poor model,

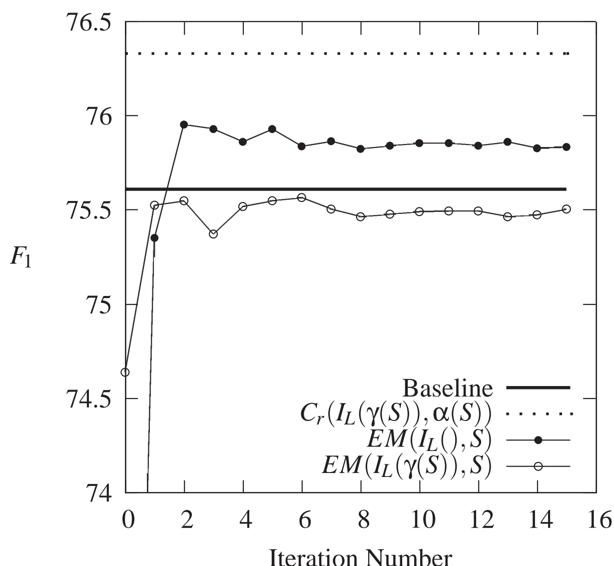

Figure 5.3: Performance over S for C_r and EM.

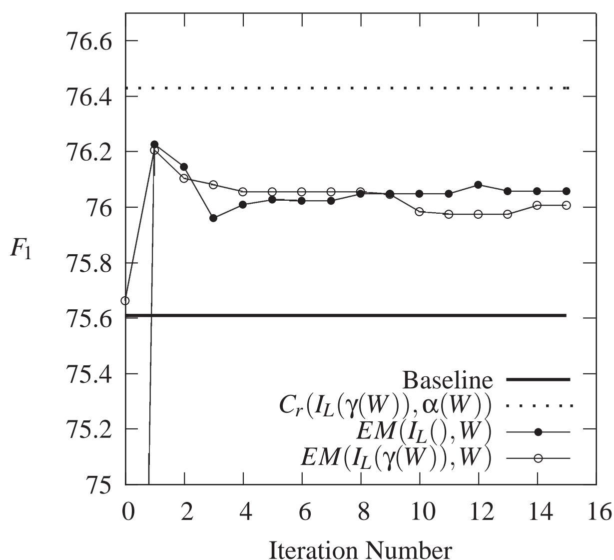

Figure 5.4: Performance over W for C_r and EM.

such as that built from S (Figure 5.3), a 'classical' pattern emerges with relatively steady improvement from iteration 1 until performance asymptotes. However, when the starting point is better (e.g. in Figures 5.4 and 5.5), the 'initial' pattern emerges. That is, the best performance is reached after a small number of iterations.

Domain Adaptation

When building NLP applications it is preferable to accurately tune a parser to a new domain with minimal manual effort. To obtain training data in a new domain, annotating a corpus with partial-bracketing information is much cheaper than full annotation. To investigate whether such data would be of value, we considered W to be the corpus over which we were tuning and applied the best performing model trained over S, $C_r(I_L(\gamma(S)), \alpha(S))$, as our initial model. That

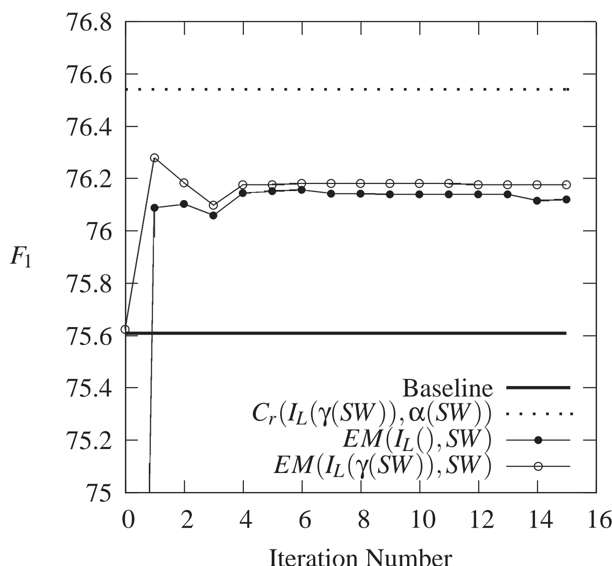

Figure 5.5: Performance over *SW* for C_r and EM.

is, we consider W as in-domain data (as DepBank is extracted from section 23 of the WSJ) and we utilise a model trained over out-of-domain data to annotate this corpus. We consider this a semisupervised domain adaptation task.

Figure 5.6 illustrates the performance of C_r compared to EM. Tuning using C_r was not significantly different from the model built directly from the entire data set with C_r, achieving 76.57% as opposed to 76.54% F_1. In contrast, EM performs better given in-domain data from the beginning rather than tuning to the new domain. C_r outperforms EM except for one particular EM iteration as shown in this figure. Though it is worth noting the behaviour of EM given only the tuning data (W) rather than the data from both domains (SW). In this case, the graph illustrates a combination of Elworthy's 'initial' and 'classical' patterns. The steep drop in performance (to under 70% F_1) after the first iteration is probably due to loss of information from S. However, this run also eventually converges to similar performance, suggesting that the information in S is effectively disregarded as it forms only a small portion of SW, and that these runs effectively converge to a local maximum over W.

Bacchiani *et al.* (2006) explore the effect of weighting the contribution (frequency counts) of the in-domain and out-of-domain training data sets. They demonstrates that altering this weighting can have beneficial effects. However, the *SW* corpus already contains a disproportionate number of in-domain W sentences (40K of 47K). Furthermore, our results suggest that the parsing models are effectively converging on the WSJ in either case.

5.5.2 Unsupervised Training

The confidence-based models were primarily developed for use over semisupervised corpora. However, we wished to consider the performance of these models given unsupervised data. In this case, we illustrate the approximate accuracy lower bound for these models (given an initial parsing model) as we rely entirely on the initial model to annotate the bootstrap data. In this section we first describe the experimental setup, and following, describe the performance of the different confidence measures over the unsupervised Susanne corpus S_u available in this work. Finally, we train the same parsing model using EM over the unsupervised data, and contrast the

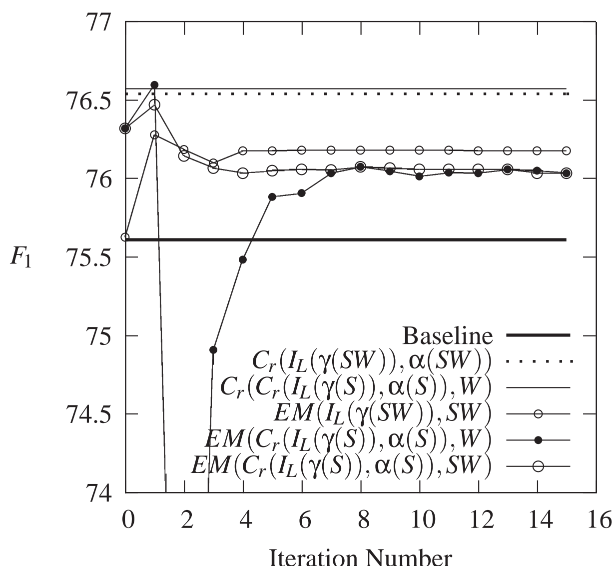

Figure 5.6: Tuning over the WSJ (W) from Susanne (S).

resulting parser's performance to that achieved by the current parser and the confidence-based methods.

Experimental Setup

We consider an initial uniform probability model, i.e. $I_L()$. We output the top 1K derivations from this initial model over the S_u corpus, and include all these derivations within the confidence-based framework.

Confidence-based Approaches

Table 5.3 illustrates the performance of the unsupervised models, where surprisingly, the C_r model achieves lower though *statistically indistinguishable* performance to that of the current, fully supervised, parsing model. The unsupervised corpus contains around 7K sentences while the supervised corpus contains 5K. The additional training data, combined with the relatively good performance of the uniform model, is able to achieve fairly impressive results. This model significantly outperforms the alternative, C_n and C_p, confidence measures, indicating that this method is fairly robust to the choice of initial parsing model. These results, combined with those discussed previously, indicate that C_r performs well over corpora with varying levels of annotation.

The results of self-training i.e. C_1 have resulted in an increase in performance over the initial model. We previously hypothesised that the marginal increase for this model over S is due to the lack of inherent bias within the initial model, as the unambiguous seed data $\gamma(S)$ was a relatively small data set. This theory is supported by these results in which the initial uniform model contains no learnt bias. Any learnt (incorrect) linguistic bias appears to be compounded in the resulting self-trained model, though this is expected as these biases are selected for when self-annotating the data. The self-training model also outperforms the C_p model over this data. However, the probability-based confidence measure is expected to perform poorly if the initial model's probability model is unable to assign probability mass to correct constituents only. In these experiments, the initial uniform model we employ does not reflect any preference between competing parse actions.

Model	Prec	Rec	F_1	$P(z)^{\ddagger}$
Baseline	77.05	74.22	75.61	-
$I_L()$	70.98	68.90	*69.92*	0.0000
$C_1(I_L(), S_u)$	74.94	72.50	*73.70*	0.0000
$C_n(I_L(), S_u)$	75.74	73.25	*74.48*	0.0024
$C_r(I_L(), S_u)$	76.28	73.81	75.02	0.1170
$C_p(I_L(), S_u)$	72.25	70.23	*71.23*	0.0000

Table 5.3: Performance of all unsupervised confidence-based models on DepBank. ‡represents the statistical significance of the system against the baseline model.

Comparison to EM

We perform unsupervised EM over S_u using the IOA$_{LR}$, as described in §4.1.3, the cross-entropy of which is shown in Figure 5.7. Figure 5.8 illustrates the performance of the unsupervised C_r and EM over S_u. The 'initial' pattern of Elworthy (1994) emerges for EM, where the best performing iterations are those from 2 to 4, where the EM outperforms the confidence-based method for these iterations. These iterations of EM, and the confidence-based method C_r, achieve lower performance compared to the current model, though these results are not significant. However, if we were unable to manually select the best model (iteration) for EM, then the resulting EM model (the model resulting for the converged fifth iteration) would be significantly lower (z-value of 0.0040) to that of the current model. Therefore the C_r training method is preferable, even in an unsupervised training task.

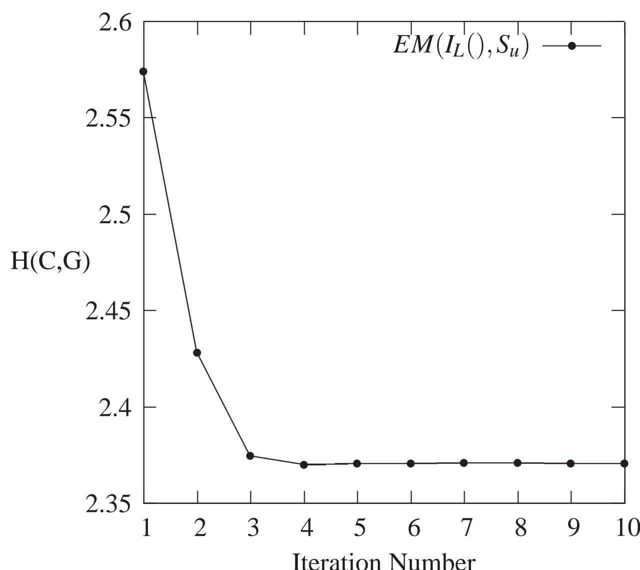

Figure 5.7: Cross entropy convergence for EM over unsupervised S_u.

5.6 Discussion

We have presented several semisupervised *confidence-based* training methods which have significantly improved performance over the current supervised method, while also reducing the

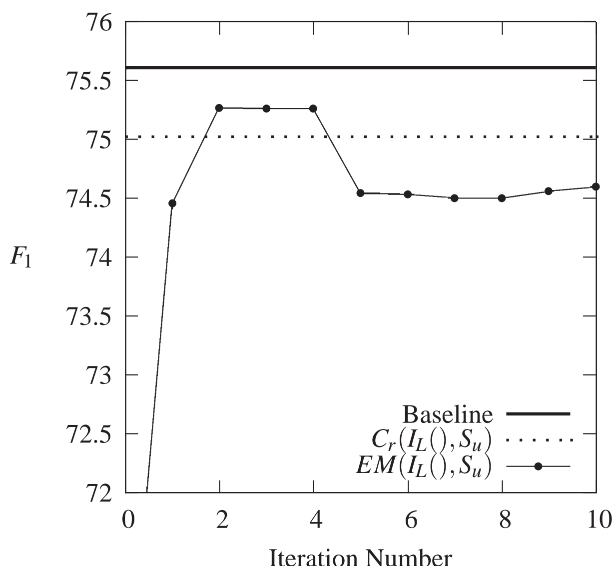

Figure 5.8: Performance over S_u for C_r and EM.

manual effort required to create training or tuning data. We have shown that given a medium-sized unlabelled partially-bracketed corpus, the confidence-based models achieve superior results to those achieved with EM applied to the same SGLR parse selection model. Indeed, a bracketed corpus provides flexibility as existing treebanks can be utilised despite the incompatibility between the system grammar and the underlying grammar of the treebank. Mapping an incompatible annotated treebank to a compatible partially-bracketed corpus is relatively easy compared to mapping to a compatible fully-annotated corpus.

An immediate benefit of this work is that (re)training parsers with incrementally-modified grammars based on different linguistic frameworks should be much more straightforward. For example, see Oepen *et al.* (2002) for a discussion of the problem. Furthermore, our findings suggest that it may be possible to usefully tune a parser to a new domain with less annotation effort.

Of the confidence measures considered, C_r consistently performed the best, illustrating its robust nature across different domains and varying levels of annotation. The 'self-training' C_1 measure performed poorly in the semisupervised training task for several corpora, supporting the findings in initial self-training studies.

Our findings mirror those of Elworthy (1994) and Merialdo (1994) for POS tagging and suggest that EM is not always the most suitable training method (especially when some in-domain training data is available). The confidence-based methods were successful because the level of noise introduced did not outweigh the benefit of incorporating all derivations compatible with the bracketing in which the derivations contained a high proportion of correct constituents. These findings may not hold if the level of bracketing available does not adequately constrain the derivations considered. Hwa (1999) describes a related investigation with EM. However, we illustrated that even over an unsupervised corpus, the C_r confidence method achieved statistically equivalent performance to the extant model. Although, this may not translate to other parsers in which the uniform statistical model performs poorly or if the models are also required to infer the grammar during training.

In future work we intend to further investigate the problem of tuning to a new domain, given

that minimal manual effort is a major priority. We hope to develop methods which required no manual annotation. For example, high precision automatic partial bracketing using phrase chunking and/or named entity recognition techniques might yield enough information to support the training methods developed in this work.

Finally, further experiments regarding alternative confidence measures within the framework described may prove beneficial. For example, we could normalise the ranking measure based on the number of consistent derivations. This confidence measure would ensure that each sentence in the bootstrap data contributes equally to the LR parse action frequency counts. Several other variants of the framework may also improve the final parse model's performance. For example, we could perform iterative rounds of reannotation over portions of the bootstrap data, as applied in the previous self-training experiments, for example those described by Steedman *et al.* (2003b).

Chapter 6

Conclusion

The focus of this thesis was the optimisation, in terms of both parser accuracy and efficiency, of an extant and well-developed SGLR parser. In this chapter we review the novel contributions of this thesis and also describe future lines of investigation. This discussion is organised by chapter.

Chapter 3 considered the optimal choice of PoS tag model employed by the extant parser, given that a front-end PoS tagger (i.e. preprocessing component) is applied. Previous work shows that parser efficiency improves if tag ambiguity is resolved by the front-end PoS tagger, though the accuracy and coverage of the parser declines as the level of tag error increases. Consequently, we investigated the optimum level of tag ambiguity to pass to the parser considering both PoS tag and parser performance. As far as the author is aware, this work is the first to perform such a broad comparison. This broad comparison is important as different tag confusions are not equally detrimental to parser output (illustrated within the experimental results of this chapter). While the initial tag selection models investigated achieve poor tagging performance, we show that gains in parser accuracy and coverage are available if we allow the parser to resolve some of the tag ambiguity. However, this results in a significant decline in parser efficiency.

Parsing results suggested that tag errors introduced by the PoS tagger cause a large proportion of the resulting fragmentary parses found over the single tag per word input. We hypothesised that a grammar (especially one that is well-constrained over the terminals) may be relied upon to find a parse over correct PoS tag sequences only. In response, we described a *dynamic* tag selection model similar to that applied by Clark & Curran 2004a. This model increases the number of tags considered in parsing, starting from the set of most probable tags, until a complete derivation is found. Here, the known trade-off between parse ambiguity and PoS tag error provides a means to gauge PoS tag error based on parser output. An artificial implementation of the model achieves the parsing accuracy and efficiency for the large proportion of nonfragmentary parses over the single tpw input. However, it also improves the accuracy over the remaining set of (otherwise fragmentary) parses, as we reparse these with larger tag sets (removing the tag errors introduced) resulting in an overall increase in parser accuracy and coverage. As we only reparse a relatively small proportion of sentences our efficiency is improved over that of the parser considering multiple tag per word input across all test sentences.

In future work, we aim to implement the dynamic model within the extant parser, as our experimental results were achieved by merging resulting parse output files only. Furthermore, we hope to apply this tag selection model over domains in which the PoS tagger achieves poor

single tpw tagging accuracy. Here, we hope to investigate whether the dynamic model can improve parse accuracy and coverage while increasing parser throughput over a range of domains. That is, whether the grammar can reliably be used to indicate the presence of tag errors over out-of-domain (tag and parser) data.

Chapter 4 defined a novel method that improves the throughput and accuracy of the 'weighted GRs' output format. The extant method required a number of processing stages to determine this output format: unpacking the n-best derivations from the parse forest, deriving the corresponding n-best GR sets and finding the unique set of GRs and corresponding weights. Although the accuracy of the output improves as the size of the n-best list considered increases, the efficiency declines in turn.

We illustrated how to obviate the need to trade off efficiency and accuracy by extracting weighted GRs directly from the parse forest using a dynamic programming approach based on the IOA. However, this method correctly calculates the weights of this output only if a *single* lexical head is found for each node in the parse forest. Related work enforces this condition by placing extra constraints on which nodes can be packed, leading to less compact parse forests. Instead, we defined a novel dynamic programming approach, the 'EWG' algorithm to enable *multiple* inside and outside probabilities for each node in the parse forest, one for each possible lexical head. Experimental results demonstrated that the novel EWG algorithm achieved substantial increases in parser accuracy and throughput for weighted GR output. EWG is available for use within the second release of RASP (see Briscoe *et al.* 2006), as an alternative method to calculate the weighted GR output format. This algorithm could be applied to any graph-structured data structure, over which we aim to estimate weighted frequency estimates for node attributes for which more than one value may apply.

We employed the parse selection strategy defined by Clark & Curran (2004b). This method applied the EWG algorithm and achieved 3.01% relative reduction in error for F_1. However, it is infeasible to define some GRs within the mapping from local-trees to GRs in RASP's grammar. Therefore each of the n-best GR sets is consistent, though may not represent a complete GR set. These 'missing' GRs do not appear within the weighted GR output and should be inferred given an incomplete, though consistent, set of GRs. The F_1 upper bound of the task is calculated using the high precision and recall GR sets determined from the weighted GR output. This upper bound is currently in the low 80's, and reflects the short-fall in the GR representation.

In future work, we aim to develop parse selection strategies directly over the weighted GR output format, the GRs of which form a directed graph (DG). Nodes of the DG are words of the input, while edges from head to dependent are labelled with the GR's type and weight. Given a (nontrivial) definition of GR consistency, we aim to determine the set of consistent GRs in this DG. For example, we could employ a suitable search algorithm (defined in graph-theory) to select the most probable consistent subset of this DG. If this work proves feasible, we aim to infer the GRs that are missing from this consistent set, to form a consistent *and* complete GR set.

Finally, in **Chapter 5**, we described a novel training framework similar to the self-training approaches employed in the literature, that can be applied to a wide range of parsing models. The framework considered weighting the contribution of all derivations within the n-best list that are *consistent* with the training corpus annotation (if any). This weighting is based on the corresponding *derivation confidence* of the initial parsing model. We described a number of different confidence measures, and compared these over a range of different domains. The C_r measure, based on the inverse of the derivation's rank, proved to be fairly robust in terms of the

choice of initial parsing model, training domain and the level of corpus annotation available.

The confidence-based parsing models consistently outperformed Expectation-Maximisation in the experimentation described, over both semisupervised and unsupervised training corpora. Furthermore, constraining the confidence-based models using unlabelled partially-bracketed data (automatically extracted from existing corpora) resulted in several parsing models that significantly outperformed the extant parser. The C_r model, trained over the bracketed Susanne corpus S, has been adopted as the training method within the extant parser. This model improves the accuracy of the resulting parser, moreover, it aids in grammar development as the grammar writer is no longer required to maintain a fully-supervised treebank. In fact, this method requires no manual effort on behalf of the grammar writer and the extant parser's training method is now fully automated.

In future work, we aim to investigate methods to usefully tune the parser to new domains, given that several studies have illustrated that even limited in-domain training data can significantly improve parser accuracy. The unsupervised training experiments illustrated that the C_r method may be sufficiently robust to handle unsupervised domain adaptation. We also aim to investigate automatic methods (requiring no direct annotation) for providing partial-annotation to constrain the set of derivations considered. Here, high precision automatic partial bracketing using phrase chunking and/or named entity recognition techniques might yield enough information to support the training methods developed in this thesis.

References

AHO, A., SETHI, R. & ULLMAN, J. (1986). *Compilers: principles, techniques, and tools*. Addison-Wesley, Boston, MA.

ALSHAWI, H., ed. (1992). *The Core Language Engine*. MIT Press, Cambridge, MA.

ANDERSON, T., EVE, J. & HORNING, J. (1973). Efficient LR(1) parsers. *Acta Informatica*, 2(1), 12–39.

ASTON, G. & BURNARD, L. (1998). *The BNC Handbook*. Edinburgh University Press, Edinburgh.

BACCHIANI, M., RILEY, M., ROARK, B. & SPROAT, R. (2006). MAP adaptation of stochastic grammars. *Computer Speech and Language*, 20(1), 41–68.

BAKER, J. (1979). Trainable grammars for speech recognition. In *Proceedings of the Spring Conference of the Acoustical Society of America*, 547–550, Boston, MA.

BIKEL, D. (2004). Intricacies of Collins' parsing model. *Computational Linguistics*, 30(4), 479–511.

BLACK, E., ABNEY, S., FLICKENGER, D., GDANIEC, C., GRISHMAN, R., HARRISON, R., HINDLE, D., INGRIA, R., JELINEK, F., KLAVANS, J., LIBERMAN, M., MARCUS, M., ROUKOS, S., SANTORINI, B. & STRZALKOWSKI, T. (1991). A procedure for quantitavely comparing the syntactic coverage of English grammars. In *Proceedings of the Fourth DARPA Workshop on Speech and Natural Language*, 306–311, Morgan Kaufman, San Mateo, CA.

BOD, R. (1998). *Beyond Grammar: An Experience-Based Theory of Language*. CSLI Publications, Cambridge University Press.

BOD, R. (2006). An all-subtrees approach to unsupervised parsing. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, 865–872, Sydney, Australia.

BRISCOE, E.J. (2006). *An introduction to tag sequence grammars and the RASP system parser*. Technical Report 662, Computer Laboratory, University of Cambridge.

BRISCOE, E.J. & CARROLL, J. (1993). Generalised probabilistic LR parsing of natural language (corpora) with unification-based grammars. *Computational Linguistics*, 19(1), 25–59.

BRISCOE, E.J. & CARROLL, J. (2002). Robust accurate statistical annotation of general text. In *Proceedings of the 3rd International Conference on Language Resources and Evaluation*, 1499–1504, Las Palmas, Gran Canaria.

BRISCOE, E.J. & CARROLL, J. (2006). Evaluating the speed and accuracy of a domain-independent stastistical parser on the PARC Depbank. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics Main Conference Poster Sessions*, 41–48, Sydney, Australia.

BRISCOE, E.J., CARROLL, J., GRAHAM, J. & COPESTAKE, A. (2002). Relational evaluation schemes. In *Proceedings of the Beyond PARSEVAL Workshop at the 3rd International Conference on Language Resources and Evaluation*, 4–8, Las Palmas, Gran Canaria.

BRISCOE, E.J., CARROLL, J. & WATSON, R. (2006). The second release of the RASP system. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics Interactive Presentation Sessions*, 77–80, Sydney, Australia.

CARROLL, J. (1993). *Practical unification-based parsing of natural language*. Technical Report 314, Computer Laboratory, University of Cambridge.

CARROLL, J. & BRISCOE, E.J. (2002). High precision extraction of grammatical relations. In *Proceedings of the 19th International Conference on Computational Linguistics*, 134–140, Taipei, Taiwan.

CARROLL, J., BRISCOE, E.J. & SANFILIPPO, A. (1998). Parser evaluation: a survey and a new proposal. In *Proceedings of the Workshop on The Evaluation of Parsing Systems at the 1st International Conference on Language Resources and Evaluation*, 447–454, Granada, Spain.

CHARNIAK, E. (1994). *Statistical Language Learning*. MIT Press, Cambridge, MA.

CHARNIAK, E. (1997). Statistical parsing with a context-free grammar and word statistics. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference*, 598–603, Providence, Rhode Island.

CHARNIAK, E. (2000). A maximum-entropy-inspired parser. In *Proceedings of the first conference on North American chapter of the Association for Computational Linguistics*, 132–139, Seattle, Washington.

CHARNIAK, E., CARROLL, G., ADCOCK, J., CASSANDRA, A., GOTOH, Y., KATZ, J., LITTMAN, M. & MCCANN, J. (1996). Taggers for parsers. *Artificial Intelligence*, 85(1–2), 45–57.

CHARNIAK, E. & JOHNSON, M. (2005). Coarse-to-fine n-best parsing and MaxEnt discriminative reranking. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, 173–180, Ann Arbor, Michigan.

CLARK, S. & CURRAN, J. (2004a). The importance of supertagging for wide-coverage CCG parsing. In *Proceedings of the 20th International Conference on Computational Linguistics*, 282–288, Geneva, Switzerland.

CLARK, S. & CURRAN, J. (2004b). Parsing the WSJ using CCG and log-linear models. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics*, 104–111, Barcelona, Spain.

CLARK, S. & CURRAN, J. (2006). Partial training for a lexicalised-grammar parser. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, 144–151, New York, New York.

CLARK, S., CURRAN, J. & OSBORNE, M. (2003). Bootstrapping PoS taggers using unlabelled data. In *Proceedings of the 7th conference on Natural language learning at Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, 49–55, Edmonton, Canada.

COHN, D.A., ATLAS, L. & LADNER, R.E. (1994). Improving generalization with active learning. *Machine Learning*, 15(2), 201–221.

COLLINS, M. (1999). *Head-Driven Statistical Models for Natural Language Parsing*. Ph.D. thesis, University of Pennsylvania.

COLLINS, M. (2004). Parameter estimation for statistical parsing models: Theory and practice of distribution-free methods. In H. Bunt, J. Carroll & G. Satta, eds., *New developments in Parsing Technology*, 19–55, Kluwer Academic Publishers, Norwell, MA.

COLLINS, M. & DUFFY, N. (2002). New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, 263–270, Philadelphia, Pennsylvania.

COLLINS, M. & ROARK, B. (2004). Incremental parsing with the perceptron algorithm. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, 111–118, Barcelona, Spain.

COPESTAKE, A. (2003). *Report on the Design of RMRS*. Technical Report D1.1a, Computer Laboratory, University of Cambridge.

DALRYMPLE, M. (2006). How much can part-of-speech tagging help parsing? *Natural Language Engineering*, 12(4), 373–389.

ELWORTHY, D. (1994). Does baum-welch re-estimation help taggers? In *Proceedings of the 4th Conference on Applied Natural Language Processing*, 53–58, Stuttgart, Germany.

GILDEA, D. (2001). Corpus variation and parser performance. In *Proceedings of the 6th conference on Empirical Methods in Natural Language Processing*, 167–202, Pittsburgh, PA.

GROVER, C., CARROLL, J. & BRISCOE, E.J. (1993). *The Alvey Natural Language Tools grammar (4th Release)*. Technical Report 284, Computer Laboratory, University of Cambridge.

HOCKENMAIER, J. (2003). *Data and models for statistical parsing with Combinatory Categorial Grammar*. Ph.D. thesis, School of Informatics, The University of Edinburgh.

Hwa, R. (1999). Supervised grammar induction using training data with limited constituent information. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, 73–79, College Park, Maryland.

Inui, K., Sornlertlamvanich, V., Tanaka, H. & Tokunaga, T. (1997). A new formalization of probabilistic GLR parsing. In *Proceedings of the 5th International Workshop on Parsing Technologies*, 123–134, Cambridge, MA.

Johansson, S., Atwell, R., Garside, R. & Leech, G. (1986). *The tagged LOB corpus: user's manual*. Norwegian Computing Centre for the Humanities, Bergen.

Jurafsky, D. & Martin, J. (2000). *Speech and Language Processing*. Prentice-Hall, Upper Saddle River, NJ, USA.

Kaplan, R. & King, T. (2003). Low-level mark-up and large-scale LFG grammar processing. In *Proceedings of the 8th International Lexical Functional Grammar Conference*, 238–249, Albany, NY.

Kaplan, R., Riezler, S., King, T., Maxwell, J., Vasserman, A. & Crouch, R. (2004). Speed and accuracy in shallow and deep stochastic parsing. In *Proceedings of the Human Language Technology Conference and the 4th Annual Meeting of the North American Chapter of the Association for Computational Linguistics*, 97–104, Boston, MA.

King, T., Crouch, R., Riezler, S., Dalrymple, M. & Kaplan, R. (2003). The PARC 700 Dependency Bank. In *Proceedings of the 4th International Workshop on Linguistically Interpreted Corpora at the 10th Conference of the European Chapter of the Association for Computational Linguistics*, 1–8, Budapest, Hungary.

Kipps, J. (1989). Analysis of Tomita's algorithm for general context-free parsing. In *Proceedings of the 1st International Workshop on Parsing Technologies*, 193–202, Pittsburgh, PA.

Klein, D. & Manning, C. (2002). A generative constituent-context model for improved grammar induction. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, 128–135, Philadelphia, PA.

Klein, D. & Manning, C. (2003). Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, 423–430, Sapporo, Japan.

Klein, D. & Manning, C. (2004). Corpus-based induction of syntactic structure: models of dependency and constituency. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, 478–485, Barcelona, Spain.

Kristensen, B.B. & Madsen, O.L. (1981). Methods for computing LALR(k) lookahead. *ACM Transactions on Programming Languages and Systems*, 3(1), 60–82.

Kudo, T., Suzuki, J. & Isozaki, H. (2005). Boosting-based parse reranking with subtree features. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, 189–196, Ann Arbor, Michigan.

Lari, K. & Young, S. (1990). The estimation of stochastic context-free grammars using the Inside-Outside algorithm. *Computer Speech and Language*, 4(1), 35–56.

LIN, D. (1998). Dependency-based evaluation of MINIPAR. In *Proceedings of the Workshop on The Evaluation of Parsing Systems at the 1st International Conference on Language Resources and Evaluation*, Granada, Spain.

MARCUS, M.P., SANTORINI, B. & MARCINKIEWICZ, M.A. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2), 313–330.

MCCLOSKY, D., CHARNIAK, E. & JOHNSON, M. (2006). Effective self-training for parsing. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, 152–159, New York, New York.

MERIALDO, B. (1994). Tagging English Text with a probabilistic model. *Computational Linguistics*, 20(2), 155–171.

MIYAO, Y. & TSUJII, J. (2002). Maximum entropy estimation for feature forests. In *Proceedings of the Human Language Technology Conference*, 292–297, San Diego, California.

MIYAO, Y. & TSUJII, J. (2005). Probabilistic disambiguation models for wide-coverage HPSG parsing. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, 83–90, Ann Arbor, Michigan.

OEPEN, S. & CARROLL, J. (2000). Ambiguity packing in constraint-based parsing: practical results. In *Proceedings of the 1st conference on North American chapter of the Association for Computational Linguistics*, 162–169, Seattle, WA.

OEPEN, S., TOUTANOVA, K., SHIEBER, C., MANNING, C., FLICKINGER, D. & BRANTS, T. (2002). The LinGO Redwoods Treebank: Motivation and preliminary applications. In *Proceedings of the 19th International Conference on Computational Linguistics*, 1–5, Taipei, Taiwan.

OSBORNE, M. & BALDRIDGE, J. (2004). Ensemble-based active learning for parse selection. In *Proceedings of the North American Chapter of the Association for Computational Linguistics*, 89–96, Boston, MA.

PEREIRA, F. & SCHABES, Y. (1992). Inside-outside reestimation from partially bracketed corpora. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics*, 128–135, Newark, Delaware.

PIANO, L. (1996). *Adaptation of Acquilex tagger to unknown words – release 2*, University of Cambridge Computer Laboratory, unpublished memo.

PIERCE, D. & CARDIE, C. (2001). Limitations of co-training for natural language learning from large datasets. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 1–9, Philadelphia, PA.

PRESCHER, D. (2001). Inside-outside estimation meets dynamic EM. In *Proceedings of the 7th International Workshop on Parsing Technologies*, 241–244, Beijing, China.

RATNAPARKHI, A. (1999). Learning to parse natural language with maximum entropy models. *Machine Learning*, 34(1–3), 151–175.

RIEZLER, S., KING, T., KAPLAN, R., CROUCH, R., MAXWELL, J. & JOHNSON, M. (2002). Parsing the Wall Street Journal using a lexical-functional grammar and discriminative estimation techniques. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, 271–278, Philadelphia, PA.

SAMPSON, G. (1995). *English for the Computer*. Oxford University Press, Oxford, UK.

SARKAR, A. (2001). Applying cotraining methods to statistical parsing. In *Proceedings of the 2nd meeting of the North American Chapter of the Association for Computational Linguistics*, 1–8, Pittsburgh, PA.

SCHMID, H. & ROOTH, M. (2001). Parse forest computation of expected governors. In *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics*, 458–465, Toulouse, France.

SHARMAN, R.A. (1990). *Hidden Markov Model Methods for Word Tagging*. Technical Report 214, IBM UK Scientific Centre, Winchester, England.

SHEN, L. & JOSHI, A. (2004). Flexible margin selection for reranking with full pairwise samples. In *Proceedings of the 1st International Joint Conference of Natural Language Processing*, 446–455, Hainan Island, China.

SIEGEL, S. & CASTELLAN, N.J. (1988). *Nonparametric Statistics for the Behavioural Sciences, 2nd edition*. McGraw Hill, New York.

STEEDMAN, M., HWA, R., CLARK, S., OSBORNE, M., SARKAR, A., HOCKENMAIER, J., RUHLEN, P., BAKER, S. & CRIM, J. (2003a). Example selection for bootstrapping statistical parsers. In *Proceedings of the Annual Meeting of the North American chapter of the Association for Computational Linguistics*, 157–164, Edmonton, Canada.

STEEDMAN, M., OSBORNE, M., SARKAR, A., CLARK, S., HWA, R., HOCKENMAIER, J., RUHLEN, P., BAKER, S. & CRIM, J. (2003b). Bootstrapping statistical parsers from small datasets. In *Proceedings of the 10th Conference of the European Chapter of the Association for Computational Linguistics*, 331–338, Budapest, Hungary.

SU, K., WANG, J., SU, M. & CHANG, J. (1991). GLR parsing with scoring. In M. Tomita, ed., *Generalized LR Parsing*, 93–112, Kluwer Academic Publishers, Boston, MA.

TADAYOSHI, H., MIYAO, Y. & TSUJII, J. (2005). Adapting a probabilistic disambiguation model of an HPSG parser to a new domain. In *Proceedings of the Second International Joint Conference on Natural Language Processing*, 199–210, Jeju Island, Korea.

TOMITA, M. (1987). An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13(1-2), 31–46.

WATSON, R. (2006). Part-of-speech models for parsing. In *Proceedings of the 9th Annual CLUK Research Colloquium*, Open University, Milton Keynes.

WATSON, R. & BRISCOE, E.J. (2007). Adapting the RASP system for the CoNLL07 domain-adaptation task. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, 1170–1174, Prague, Czech Republic.

WATSON, R., BRISCOE, E.J. & CARROLL, J. (2007). Semi-supervised training of a statistical parser from unlabeled partially-bracketed data. In *Proceedings of the 10th International Workshop on Parsing Technologies*, 23–32, Prague, Czech Republic.

WATSON, R., CARROLL, J. & BRISCOE, E.J. (2005). Efficient extraction of grammatical relations. In *Proceedings of the 9th International Workshop on Parsing Technologies*, 160–170, Vancouver, Canada.

WEISCHEDEL, R., SCHWARTZ, R., PALMUCCI, J., METEER, M. & RAMSHAW, L. (1993). Coping with ambiguity and unknown words through probabilistic models. *Computational Linguistics*, 19(2), 361–382.

WRIGHT, J. & WRIGLEY, E. (1989). Probabilistic LR parsing for speech recognition. In *Proceedings of the 1st International Workshop on Parsing Technologies*, 193–202, Pittsburgh, PA.