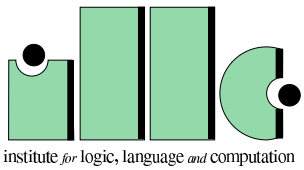
Investigations in Logic, Language and Computation

ILLC Dissertation Series 1995-19



For further information about ILLC-publications, please contact

Institute for Logic, Language and Computation
Universiteit van Amsterdam
Plantage Muidergracht 24
1018 TV Amsterdam
phone: +31-20-5256090
fax: +31-20-5255101
e-mail: illc@fwi.uva.nl

Investigations in Logic, Language and Computation

Research op de raakvlakken van logica, taal en berekening

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit Utrecht
op gezag van de Rector Magnificus, Prof. dr. J. A. van Ginkel
ingevolge het besluit van het College van Decanen
in het openbaar te verdedigen
op maandag 4 december 1995 des ochtends te 10.30 uur

door

Henricus Marinus Franciscus Maria Aarts

geboren op 25 mei 1965 te Berlicum

Promotor: Prof. dr. Michael J. Moortgat
Onderzoeksinstituut voor Taal en Spraak
Universiteit Utrecht

Co-promotor: Dr. Theo M. V. Janssen
Faculteit Wiskunde en Informatica
Universiteit van Amsterdam

The investigations were supported by project NF 102/62-356 ('Structural and Semantic Parallels in Natural Languages and Programming Languages'), funded by the Netherlands Organization for Scientific Research (NWO).

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Aarts, Henricus Marinus Franciscus Maria

Investigations in logic, language and computation /
Henricus Marinus Franciscus Maria Aarts. - Amsterdam :
Institute for Logic, Language and Computation,
Universiteit van Amsterdam. - Ill. - (ILLC dissertation
series ; 1995-19)
Proefschrift Universiteit van Amsterdam. - Met lit. opg. -
Met samenvatting in het Nederlands.
ISBN 90-74795-39-0
NUGI 855
Trefw.: Prolog (programmeertaal) / complexiteit /
computerlinguïstiek.

Copyright © 1995 by Erik Aarts, Amsterdam

Cover design by the author. Typeset by the author with LaTeX in the font *New Century Schoolbook*. Printed and bound by Ponsen en Looijen BV, Wageningen.
The e-mail address of the author is `aarts@fwi.uva.nl`. His WWW home page is `http://www.fwi.uva.nl/~aarts/`

Contents

Acknowledgments **7**

Prologue **9**

I Grammar Formalisms **11**

1 Introduction **13**
1.1 Categorial Grammars 14
1.2 Context-sensitive Grammars 16

2 The Non-associative Fragment of L **19**
2.1 Categorial Grammars 19
2.2 Recognition for Categorial Grammars Based on NL 25

3 The Second Order Fragment of L **29**
3.1 Categorial Grammars 29
3.2 The System Aux 31
3.3 Cut Elimination in Aux 34
3.4 The System ApplComp 36
3.5 The Algorithm 38
3.6 Proof of Correctness of the Algorithm 41
3.7 Discussion 42

4 Acyclic Context-sensitive Grammars **45**
4.1 Definitions 46
4.1.1 Context-sensitive Grammars 46
4.1.2 Labeled Context-sensitive Grammars 47
4.1.3 Acyclic Context-sensitive Grammars 47
4.1.4 Growing Context-sensitive Grammars 48
4.2 An Example 48
4.3 Properties of Acyclic Context-sensitive Grammars 49
4.3.1 Generative Power of Acyclic CSG's 49
4.3.2 Complexity of Acyclic CSG's 51
4.4 Uniform Recognition for ACSG is NP-complete 53
4.5 Discussion 55

II Programming in Logic **59**

5 Complexity of Prolog Programs **61**
5.1 Introduction 61
5.2 Other Research 63
5.3 Prolog 64
5.3.1 Introduction 64
5.3.2 Definitions 65
5.4 Space Complexity 73
5.5 Time Complexity 77
5.6 Well-moded and Nicely Moded Programs 79
5.7 A Lower Estimate 82
5.8 A Small Recapitulation 83
5.9 Two Examples 84
5.10 Meta-logical Predicates 87
5.11 Further Research 89
5.12 Existing Implementations of Earley Interpreters 90

6 Proof of the Time Complexity Result **91**
6.1 A More Efficient Earley Interpreter 91
6.2 Complexity of the Earley interpreter 97
6.3 Improved Earley Deduction 100

7 Fragments of L in Prolog **103**
7.1 Non-associative Lambek Grammar 103
7.1.1 Implementation 103
7.1.2 Complexity Analysis 104
7.2 Second Order Lambek Grammar 106
7.2.1 Implementation 106
7.2.2 Complexity Analysis 109

A An Implementation of the Earley Interpreter in Prolog **113**

B A Reduction of 3-SAT to ACSG Recognition **119**
B.1 The Reduction 119
B.2 A Derivation 121

Bibliography **123**

Samenvatting **127**

Summary **129**

Curriculum Vitae **131**

Acknowledgments

Research is sometimes an individual enterprise and sometimes it is not. A lot of people contributed to the research presented in this dissertation.

First I want to thank Michael Moortgat and Theo Janssen.

Michael gave me the freedom to work on the topics that interested me. Unfortunately, the problems that interested us appeared to be hard problems that often were not solved. Theo stimulated me to generate articles in proceedings and journals, knowing that this would eventually lead to a PhD thesis. Because I shared a room with Theo there was nothing I could hide from him and this has been very important.

Pieter Uit den Boogaart raised my interest in computational linguistics when I was a student at the University of Technology in Eindhoven. Harry Bunt and his group at Tilburg University took care of my education in the field. Once you take a course given by Jan van Eijck there is no turning back. Everything was good in Tilburg, so I didn't hesitate a moment when I was asked to stay in Tilburg after my graduation. I started working on the complexity of grammars that allow tangling structures (DPSG). Some of the results obtained there are in this dissertation. I learned a lot from my colleagues at the ITK and had a good time there, especially with my roommate Joop van Gent.

In 1991 I joined the NWO project 'Structural and Semantic Parallels in Natural Languages and Programming Languages', the //-project for short. Participation in this project was very stimulating. Guests were invited from abroad and there was an international workshop organized by the //-project every year. Especially the workshop in January 1993 'Grammar, Proof Theory and Complexity' which I helped to organize, was very important for me. Although a lot of communication goes via E-mail in the scientific community nowadays, meeting colleagues in real life is crucial. The //-project enabled me to do this. Furthermore I want to thank Aravind Joshi who invited me to the Institute for Research in Cognitive Science in Philadelphia. This visit was also of great influence, although I haven't been able to show a clear connection between Tree Adjoining Grammars and Categorial Grammars.

I want to thank the following people for the discussions and the scientific contact I had with them: Johan van Benthem, Peter van Emde Boas, Annius Groenink, Maciej Kandulski, Wessel Kraaij, Natasha Kurtonina, Alain Lecomte, Glyn Morill,

Mati Pentus, Fernando Pereira, Khalil Simáan, Mark Steedman, Leen Torenvliet, Marten Trautwein, Kees Vermeulen, Koen Versmissen, Eelco Visser, Vijay-Shanker, David Weir, and of course Kees Trautwein who has allowed me to use our joint article in this dissertation.

I want to thank the members of the graduation committee; Krzysztof Apt, Wojciech Buszkowski, Jan van Eijck, Jan Landsbergen, and Leen Torenvliet, for the work they have done and for their comments.

The sections about fragments of the Lambek calculus were greatly influenced by Wojciech Buszkowski. The section about acyclic context-sensitive grammars was influenced by Gerhard Buntrock and the Prolog section by Gertjan van Noord. These three people are VIP's in this dissertation.

What is a man without daily colleagues like Yuri Engelhardt, Sophie Fischer, Herman Hendriks, Dirk Heylen, Willem-Olaf Huijsen, Ernest Rotterdam and Martijn Spaan? Nothing.

Dirk Heylen and Marten Trautwein proofread my dissertation and found many errors. Thanks.

I would like to thank my parents and my in-laws for their support. Last but not least I want to thank Hilde. Her love and understanding were essential, just like the love and non-understanding of Els and Pleun.

Amsterdam Erik Aarts
October, 1995.

Prologue

As the title says, this PhD thesis is on the interface between logic, language and computation. The main perspective of this thesis is that of complexity theory. The merit of applying complexity theory to certain problems is that complexity analyses provide different kinds of information. Complexity analysis does not only show *how difficult* a problem is, but also *why* it is difficult, and possibly *how* we can change the problem such that it becomes less difficult.

The thesis is about the complexity of two grammar formalisms and a programming language. The grammar formalisms are *categorial grammar* based on *Lambek calculi* and a restricted form of the *context-sensitive rewrite grammars*: the *acyclic* context-sensitive grammars. The programming language is *Prolog*. Complexity theory distinguishes two types of complexity: time complexity and space complexity. Time complexity is defined as the number of steps that some machine needs to solve a problem. Space complexity is the amount of memory the machine needs. For an introduction in complexity theory see Garey and Johnson (1979). The problems we are going to analyze (i.e. estimating how much space and time it costs to solve these problems) are the following.

For the grammar formalisms we have some *generator G* that generates sentences. In categorial grammar, the generator consists of a lexicon of words associated with types, and some (Lambek based) calculus. In acyclic context-sensitive grammars, the generator consists of a context-sensitive grammar. Besides the generator we also have an input sentence. The problem we try to solve is the following: "Is the input sentence generated by the generator?".

A Prolog program consists of a number of facts and rules. The rules say how we can derive new conclusions from the given facts. The problem we will discuss in this thesis is: "Given a program with rules and facts, is conclusion C derivable?".

At first sight there seems to be no relation between the recognition problem for grammar formalisms and the derivability problem in Prolog. But there are similarities. The Prolog rules can be seen as a representation of infinitely many context-free rewrite rules. Every substitution for the variables gives another rewrite rule. The facts can be seen as infinitely many so-called epsilon-rules, i.e. they rewrite to the empty

string. Suppose we want to know whether a conclusion C is derivable from a program. We transform the program to an "infinite context-free grammar" with start symbol C. Then the derivability problem is equivalent to the question whether the empty string is grammatical. We have reduced the derivability problem to a recognition problem.

The fact that the grammar is infinite makes that we can not simply use parsing theory for Prolog theorem provers. But we can use elements of parsing theory to obtain theorem provers. The relationship between Prolog and grammars has been worked out in (Deransart and Maluszynski 1993). They show that Prolog programs are equivalent to *W-grammars*.

Organization of the thesis

The thesis consists of two parts that can be read separately: "Grammar Formalisms", and "Programming in Logic". Part I consists of the following chapters:

- Chapter 1 Introduction
- Chapter 2 Non-associative Lambek calculus NL
- Chapter 3 The second order fragment of Lambek calculus
- Chapter 4 Acyclic Context-Sensitive Grammars

Part II consists of the following chapters

- Chapter 5 Complexity of Prolog programs
- Chapter 6 Proof of the time complexity result
- Chapter 7 The link between the first part and the second part. It shows how we can use the method from chapter 5 for proving that membership for fragments of L can be computed in polynomial time.

Appendix A contains an implementation in Prolog of the efficient meta-interpreter described in chapter 6. Appendix B contains a reduction from 3SAT to ACSG.

Part I

Grammar Formalisms

Chapter 1

Introduction

This part of the thesis is about the definition of sets of strings and about algorithms that decide for such a set whether some sentence is in the set or not. The goal is to describe the syntax of natural languages (e.g. English). A natural language can be seen as an infinite set of sentences. The number of words in the language is finite. Systems that are developed to describe such languages are called *grammar formalisms*.

In the late 50's two formalisms were developed at the same time. Both formalisms try to define the set of syntactically well-formed sentences of some natural language. The first formalism is the *rewrite grammar* introduced by Chomsky (1959). The second is the *Lambek calculus*, introduced by Lambek (1958). The first method is a rewriting system, whereas Lambek's system is a logical system. The latter has a syntactic and a semantic part: the proof theory and the model theory. With some oversimplification the two methods can be explained as follows. Suppose we have the following sentences:

1. John walks.
2. The man walks.
3. The dog walks.

We see that "*John*", "*the man*" and "*the dog*" can occur in the same position in sentences. Chomsky formalized this as follows. He introduced *auxiliary symbols*, (in fact auxiliary words), e.g. the symbols S and NP, and *semi-sentences*, like "NP *walks*". To generate semi-sentences, there are rules that allow replacement of symbols by other symbols. E.g., the sentence symbol S can be replaced by "NP *walks*". The auxiliary symbol NP can be replaced by "*John*", "*the man*" or "*the dog*".

Lambek (1958) took another approach. He saw the set of words that can fill the open position in $\ldots$*walks* as just another language. We have to define a lot of languages instead of one. Therefore operations on languages are introduced: they can be "multiplied" and "divided". If a sentence is in the language X we say that the sentence has type X. E.g. we can say that *John*, *the man* and *the dog* have type np and that the three sentences have type s. Then we can "divide s by np", i.e., the sub-sentence *walks* has the type $np\backslash s$ (an s missing an np on the lefthand side). We can repeat this: if *walks* has type $np\backslash s$ and *John walks* has type s, then *John* has the type $s/(np\backslash s)$.

We can repeat this infinitely many times. Therefore *John* has infinitely many types. Lambek has shown that we can define all types of all sentences with a *lexicon* and a *calculus*. The lexicon assigns types to the words of the language. With the calculus we can *infer* the types of all sentences. Such a pair of a lexicon and a calculus is called a *categorial grammar*. This description is purely syntactic. The semantics of categorial logics is not described here. A second oversimplification is that we look at the weak generative power only. But it is also very important *what structures* are defined by the grammar. This point, the strong generative power, is neglected here. The main reason is that membership in a language can be very easily stated in complexity theory. Even stronger, in complexity theory *only* membership problems are used. It is much harder to deal with strong generative power and complexity than with weak generative capacity and complexity.

Once we have a definition of a language, either as a rewrite system or as a categorial grammar, we are going to *use* the definition. For some sentence, we want to decide whether the sentence is in a language or not. This problem is called the *recognition* problem. We are interested in the *time complexity* of this problem.

A lot of research has been done in the area of the complexity of the recognition problem for rewrite systems. A lot of results about classes of rewrite systems have been found. In this thesis we will consider a very specific kind of rewrite system: the acyclic context-sensitive grammars. These are grammars where we can replace a number of auxiliary symbols in one step by a number of other symbols. The most important advantage of this class is the possibilty of describing structure trees with crossing branches. This will be explained in detail later.

Compared to the rewrite systems, the complexity of categorial grammars based on Lambek's calculus has remained largely unexplored up till now. The results in this thesis are among the first results in a new field.

In order to put this thesis in context, I will sketch in this introduction a brief history of research in the field of Lambek based categorial grammar as well as of research in context-sensitive grammars.

1.1 Categorial Grammars

Lambek (1958) introduced the idea that we can define languages with a lexicon and an inference system called a calculus. Immediately after his proposal the question about the relation with Chomsky's work came up. People conjectured that Lambek's system could describe the same languages as Chomsky's context-free languages. This was proved in 1967 by Cohen (Cohen 1967). In 1978 it was shown by Zielonka (Zielonka 1978) that this proof contained a gap. Buszkowski (1986) showed that two fragments have context-free generative power: the nonassociative and the second order fragment. The problem for the general calculus was open for many years until it was solved recently by Pentus. He showed that Lambek's calculus and the context-free grammars indeed have the same generative power (Pentus 1993).

The progress in linguistic applications of Lambek's theory was minimal until the 80's. One of the reasons is given in the previous paragraph. It was generally felt in

the 60's and 70's that context-free power was not enough for the description of natural languages and that the system of Lambek was insufficient. Montague (1973) used some form of categorial grammar, but his work was focused on semantics rather than on syntax. Moortgat (1988) and Buszkowski, Marciszewski and van Benthem (1988) were the first steps that led to an increased interest of linguists in Lambek systems.

The growing popularity had two sources. First, Gazdar, Klein, Pullum and Sag (1985) showed that a lot of constructions in sentences that were regarded as "beyond context-free" were in fact expressable with a context-free grammar. The constructions that are used nowadays to prove that natural language is not context-free are pretty rare. On the other hand there has been a strong effort to extend the basic system of Lambek in order to get more power. The extension of the system was influenced by *modal logic* and by *linear logic*. Linear logic was introduced in 1987 by Girard. Although there had been earlier efforts to increase the generative power of Lambek-like calculi, the development of linear logic stimulated these attempts. Because of the clear correspondence between Linear Logic and Lambek's systems the extensions needed became obvious more or less. One can simply add new rules to the calculus of Lambek. With these new rules we get various new systems. These systems can be seen as inhabitants of a landscape of categorial systems. This landscape is in fact a categorial hierarchy, just like the Chomsky hierarchy for rewrite systems.

The inhabitants in this landscape can be identified as follows. The system originally introduced by Lambek is called the system L. If we add the structural rule of permutation (a standard rule in Linear Logic) we get the system LP. If we define systems by adding rules, then the base system should not be L but the nonassociative calculus NL (introduced by Lambek in 1961). If we add the structural rule of associativity to this system we obtain L. Another rule which can be added is the rule of dependency. Varying the presence of the associativity, the permutation and the dependency rule gives us 8 possible Lambek systems. A good traveller's guide to the categorial landscape is Moortgat (1995).

The complexity of the recognition problem for various systems in the categorial landscape has not been established yet. It has been shown (Kanovich 1991) that for LP the problem is NP-complete. The problem is still open for L. From the fact that the generative power of L is context-free we can deduce that the recognition problem is polynomial in the size of the input. But nothing is known about the uniform recognition problem, where the grammar is part of the input as well. In this thesis we solve the problem for two fragments, the nonassociative and the second order fragment. For these two fragments, we can prove that the recognition problem is in polynomial time. These results are new. Buszkowski (1986) proved that the non-associative Lambek categorial grammar is equivalent with context-free grammar. But the grammar that he obtains has exponential size and can not be used to show that the problem can be solved in polynomial time. Exponential time algorithms were found by (Janssen 1991, Trautwein 1991).

1.2 Context-sensitive Grammars

The context of the research in categorial grammar has been sketched. In this section we will sketch the history of research in the field of context-sensitive rewrite grammars.

The rewrite grammars introduced in (Chomsky 1959) were classified by him in the so-called Chomsky hierarchy. Grammars are of type 3 (regular), type 2 (context-free), type 1 (context-sensitive) or of type 0 (unrestricted). For context-free grammars input strings can be recognized in a time that is polynomial in the length of the input string as well as in the length of the grammar. Earley (1970) has shown a bound of $\mathcal{O}(|G|^2 n^3)$ where $|G|$ is the size of the grammar and n the length of the input string. In fact there is a better upperbound: $\mathcal{O}(|G|n^3)$ (Sippu and Soisalon-Soininen 1988). Recognition for context-sensitive grammars is harder: it is PSPACE-complete, even for some fixed grammars (Kuroda 1964) and (Karp 1972). Recognition of type 0 languages is undecidable (see e.g. (Lewis and Papadimitriou 1981)).

In this thesis we will introduce a formalism that can describe trees with *crossing branches* without immediately jumping to the full class of context-sensitive grammars. This formalism is called *acyclic context-sensitive grammar* and it is in between the context-free grammars and the context-sensitive grammars. Other formalisms that are in between these two classes are *growing* context-sensitive grammars and *Tree Adjoining Grammars*.

Growing CSG's are CSG's where the right-hand side of every rule is strictly longer than the left-hand side. Dahlhaus and Warmuth (1986) have shown that recognition for growing context-sensitive grammars is polynomial time for every fixed grammar. The question whether the uniform recognition problem (where the grammar is part of the input for the problem) is in P was posed by Dahlhaus and Warmuth (1986). It has been proved three times (independently) that this problem is NP-complete (Buntrock and Loryś 1992). The recognizing power of growing CSG's is beyond context-free. Two typical examples of growing CS languages that are not context-free are $\{a^n b^{2^n} c^n \mid n \geq 1\}$ and $\{a^n b^n c^n \mid n \geq 1\}$.

A formalism which is quite popular in computational linguistics is Tree Adjoining Grammar (TAG), proposed by (Joshi, Levy and Takahashi 1975). While rewrite grammars rewrite strings, TAG's rewrite *trees*. If only rewriting of the root and the leaves of a tree are allowed tree rewriting is equivalent to (context free) string rewriting. But in TAG's rewriting can also take place in the middle of a tree. This is called adjunction.

TAG's generate all context free languages and various non-contextfree languages, amongst them $a^n b^n c^n$ and $\{ww \mid w \in \{0,1\}^*\}$ (Joshi, Vijay-Shanker and Weir 1991, p. 40). The class of languages that can be generated by TAG's is called the class of Tree Adjoining Languages *TAL's*. Recognition is polynomial both in the size of the grammar and in the size of the input string (Schabes and Joshi 1988, p. 267).

The languages generated by growing CSG's and TAG's are incomparable. The TAL's have the *constant growth* property. This means that if the strings of the language are put in increasing order of length, then two consecutive lengths do not differ by arbitrarily large amounts. The language $\{a^n b^{2^n} c^n \mid n \geq 1\}$ is a growing CSL and does not have this property. Therefore the growing CSL's are not included in the TAL's.

The inclusion does not hold the other way either. The language $\{ww \mid w \in \{0,1\}^*\}$ is in TAL (Joshi et al. 1991, p. 40) but can (probably) not be generated by a growing CSG.

It has been shown (Joshi et al. 1991) that TAG's are equivalent with three other grammar formalisms: Linear Indexed Grammars, Head Grammars and Combinatory Categorial Grammar. We can summarize this section in the following table:

Model	Recogn. Power	Complexity of uniform recognition	Complexity of recognition for any fixed grammar
General CSG's	CSL	PSPACE-complete	PSPACE-complete
Growing CSG's	more than CFL's	NP-complete	P
TAG, CCG, LIG etc.	TAL's	P	P

We will show in chapter 4 how the acyclic context-sensitive grammars fit in this picture.

Chapter 2

The Non-associative Fragment of L

In this chapter[1] we present algorithms for the recognition of sentences generated by non-associative and second order Lambek based categorial grammars. First we will define categorial grammars. These grammars consist of a lexicon defining a type assignment to words, and a calculus defining the well-formed sequences of types. We will start with the non-associative grammar and switch to the second order grammar later. We present a new axiomatization of NL. This axiomatization enables us to give a polynomial translation into context-free grammars. After the translation we can use any context-free recognition algorithm to recognize a sentence generated by the NL based categorial grammar in polynomial time.

2.1 Categorial Grammars

A categorial grammar G is defined by a lexicon Lex and a calculus C. A (finite) set of *primitive types* Pr is given. Tp is the set of *types*. Types are constructed from primitive types by two type-forming operators: $/$ and $\backslash$, i.e., Tp is the smallest set including Pr such that if $x, y \in Tp$, then $x\backslash y, x/y \in Tp$. One member s of Pr is singled out as the *distinguished type*.

A *lexicon* Lex is a finite relation between an alphabet Σ and Tp ($Lex \subset \Sigma \times Tp$, $\Sigma \cap Tp = \emptyset$). If a lexicon Lex relates $a \in \Sigma$ with $A \in Tp (\langle a, A \rangle \in Lex)$, we say Lex *assigns* A *to* a.

STp is the set of *strings of types*. Furthermore, we define the set $BSTp$, of *bracketed strings of types* as the smallest set including Tp such that if $X, Y \in BSTp$, then $(X, Y) \in BSTp$. The *yield* of a bracketed string X is the string we get when we erase the brackets. Yields are elements of STp.

Now $L(G)$ is defined to be the set of strings $w \in \Sigma^*$, $w = a_1 \dots a_n$ such that for some bracketed string of types X, $yield(X) = A_1, \dots, A_n$, $\langle a_i, A_i \rangle \in Lex$ $(1 \leq i \leq n)$ and $X \to s$ is derivable in the calculus C.

[1] I want to thank *Mathematical Logic Quarterly* for their permission to use Aarts and Trautwein (1996).

We use NL as the calculus C. There is a number of possible axiomatizations for this calculus. We choose one given in Kandulski (1988) as our starting point. This calculus is called NLG_0 and can be found in Figure 2.1.

$$\text{[A1]}\quad x \to x \text{ , for } x \in Tp$$

$$\frac{(X,y) \to x}{X \to x/y}\ \text{[R1}'\text{]} \qquad \frac{(y,X) \to x}{X \to y\backslash x}\ \text{[R1}''\text{]}$$

$$\frac{X \to y \quad Y[x] \to z}{Y[(x/y,X)] \to z}\ \text{[R2}'\text{]} \qquad \frac{X \to y \quad Y[x] \to z}{Y[(X,y\backslash x)] \to z}\ \text{[R2}''\text{]}$$

$$\text{for } X, Y \in BSTp \text{ and } x, y, z \in Tp.$$

Figure 2.1: NLG_0

The notation $Y[x]$ (resp. $Y[X]$) is used to indicate the bracketed string of types Y, in which, on a certain place, type x (resp. bracketed string of types X) occurs.

Before we show that sentences can be recognized in polynomial time we first introduce two auxiliary calculi: NLG_0^* and NLG_0^{**}. The calculus NLG_0^* differs from NLG_0 in that the X's in the R1′ and R1″ rules are restricted: they must be in Tp instead of in $BSTp$. The calculus NLG_0^{**} differs from NLG_0^* in that the X's in the R2′ and R2″ rules must be in Tp instead of in $BSTp$.

The calculus NLG_0^{**} can be written down as follows (compared to NLG_0, X has been replaced by w):

$$\text{[A1]}\quad x \to x \text{ , for } x \in Tp$$

$$\frac{(w,y) \to x}{w \to x/y}\ \text{[R1}'\text{]} \qquad \frac{(y,w) \to x}{w \to y\backslash x}\ \text{[R1}''\text{]}$$

$$\frac{w \to y \quad Y[x] \to z}{Y[(x/y,w)] \to z}\ \text{[R2}'\text{]} \qquad \frac{w \to y \quad Y[x] \to z}{Y[(w,y\backslash x)] \to z}\ \text{[R2}''\text{]}$$

$$\text{for } Y \in BSTp \text{ and } w, x, y, z \in Tp.$$

Figure 2.2: NLG_0^{**}

In the sequel, we are going to prove that NLG_0, NLG_0^* and NLG_0^{**} are equivalent. We prove the following inclusions:

- $\mathrm{NLG}_0 \subseteq \mathrm{NLG}_0^*$ (Theorem 2.1.1)
- $\mathrm{NLG}_0^* \subseteq \mathrm{NLG}_0^{**}$ (Theorem 2.1.2)
- $\mathrm{NLG}_0^{**} \subseteq \mathrm{NLG}_0$ (Theorem 2.1.3)

2.1.1. THEOREM. *For all k, if some sequent $\Gamma \to A$ containing k slashes is derivable in NLG_0, then it is also derivable in NLG_0^*.*

Proof: We prove this theorem with strong induction on the number of slashes $(/, \backslash)$ in a sequent. The base case is $k = 0$. The sequent must be an axiom. No R1′ or R1″ rules are used so Theorem 2.1.1 holds for $k = 0$. Now we assume the induction hypothesis:

IH(k): For all $i < k$, if some sequent $\Delta \to B$ containing i slashes is derivable in NLG_0, then $\Delta \to B$ is also derivable in NLG_0^*.

We have to prove that if some sequent $\Theta \to C$ containing k slashes is derivable in NLG_0, then it is also derivable in NLG_0^*.

Assume $\Theta \to C$ is derivable in NLG_0. Then there is a proof Π of $\Theta \to C$. We will show that $\Theta \to C$ is also derivable in NLG_0^*. We consider various possibilities for the last step in the proof of Π. We assume that proofs are constructed from top to bottom, i.e., from the axioms we try to reach the conclusion. The last step in a proof is the step that proves the final conclusion.

Case 1: Π is an axiom. Then it is derivable in NLG_0^* too.
Case 2: The last step in Π is an R2 rule or an R1 rule with $X \in Tp$. The proof is easy. The premises of the last step contain fewer slashes than $\Theta \to C$. The induction hypothesis tells that the premises are derivable in NLG_0^*. Use the NLG_0^* proofs of the premises and the last rule of Π to construct a proof of $\Theta \to C$ in NLG_0^*.
Case 3: The last step in Π is some R1 rule (e.g. R1′, the other case is symmetric) with $X \in BSTp$, but $X \notin Tp$.

$$\dfrac{\begin{array}{c}\vdots\\ (X, b) \to a\end{array}}{X \to a/b}\ [\mathrm{R1}']$$

The induction hypothesis tells that $(X, b) \to a$ is derivable in NLG_0^*. Consider the proof in NLG_0^* of $(X, b) \to a$. The last step in this proof is not an R1 rule because $(X, b) \notin Tp$. If $(X, b) \to a$ has been derived with an R2 rule, then we have various possibilities.

$$\dfrac{\dfrac{\begin{array}{c}\cdots\end{array}}{(X, b) \to a}\ [\mathrm{R2}]}{X \to a/b}\ [\mathrm{R1}']$$

The type b is involved in the rule R2, i.e., $b = x/y$ or $b = y\backslash x$ or $b = w$ (case 3b), or it is not involved (case 3a).

Case 3a: If b is not involved, then R2 can be R2′ or R2″. The proof is similar for both cases. In the first case, the proof is of the form:

$$\dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots\\ X' \to y & (Z[c], b) \to a\end{array}}{(Z[(c/y, X')], b) \to a}\,[\mathrm{R2}']}{Z[(c/y, X')] \to a/b}\,[\mathrm{R1}']$$

We can transform this proof into the following proof:

$$\dfrac{\begin{array}{cc}\vdots & \dfrac{\vdots}{(Z[c], b) \to a}\,[\mathrm{R1}']\\ X' \to y & Z[c] \to a/b\end{array}}{Z[(c/y, X')] \to a/b}\,[\mathrm{R2}']$$

The sequents $Z[c] \to a/b$ and $X' \to y$ have fewer slashes than $Z[(c/y, X')] \to a/b$. The induction hypothesis tells that the two smaller sequents are provable in NLG_0^*. Therefore, $Z[(c/y, X')] \to a/b$ is also provable in NLG_0^*.

Case 3b: If b is involved in the R2 rule it must be the functor. X cannot be the functor because $X \notin Tp$. Type b is of the form $d\backslash c$.

$$\dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots\\ X \to d & c \to a\end{array}}{(X, d\backslash c) \to a}\,[\mathrm{R2}']}{X \to a/(d\backslash c)}\,[\mathrm{R1}']$$

The induction hypothesis says that $X \to d$ is derivable in NLG_0^*. Consider the proof of $X \to d$ in NLG_0^*. The last step in this proof must be an R2 step because $X \notin Tp$. The proof is of the following form:

$$\dfrac{\dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots\\ X' \to y & Z[e] \to d\end{array}}{Z[(e/y, X')] \to d}\,[\mathrm{R2}'] \qquad \begin{array}{c}\vdots\\ c \to a\end{array}}{(Z[(e/y, X')], d\backslash c) \to a}\,[\mathrm{R2}'']}{Z[(e/y, X')] \to a/(d\backslash c)}\,[\mathrm{R1}']$$

We can transform this into:

$$\dfrac{\begin{array}{cc}\vdots & \dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots\\ Z[e] \to d & c \to a\end{array}}{(Z[e], d\backslash c) \to a}\,[\mathrm{R2}'']}{Z[e] \to a/(d\backslash c)}\,[\mathrm{R1}']\\ X' \to y & \end{array}}{Z[(e/y, X')] \to a/(d\backslash c)}\,[\mathrm{R2}']$$

The induction hypothesis says that $X' \to y$ and $Z[e] \to a/(d\backslash c)$ have proofs in NLG_0^*.

Therefore $Z[(e/y, X')] \to a/(d\backslash c)$ has a proof in NLG_0^*. This completes the proof of Theorem 2.1.1. □

2.1.2. THEOREM. *For all k, if some sequent $\Gamma \to A$ containing k slashes is derivable in NLG_0^*, then it is also derivable in NLG_0^{**}.*

Proof: Proof with strong induction on the number of slashes k. The base case $k = 0$ is trivial. Now we assume the induction hypothesis:

IH(k): For all $i < k$, if some sequent $\Delta \to B$ containing i slashes is derivable in NLG_0^*, then $\Delta \to B$ is also derivable in NLG_0^{**}.

We have to prove that if some sequent $\Theta \to C$ containing k slashes is derivable in NLG_0^*, then it is also derivable in NLG_0^{**}.

Assume $\Theta \to C$ is derivable in NLG_0^*. Then there is a proof Π of $\Theta \to C$ in NLG_0^*. We will show that $\Theta \to C$ is also derivable in NLG_0^{**}. We consider various possibilities for the last step in the proof of Π.

Case 1: Π is an axiom. Then it is derivable in NLG_0^{**} too.

Case 2: The last step in Π is an R1 rule or an R2 rule with $X \in Tp$. The premises of the last step contain fewer slashes than $\Theta \to C$. The induction hypothesis tells that the premises are derivable in NLG_0^{**}. Use the NLG_0^{**} proofs of the premises and the last rule of Π to construct a proof of $\Theta \to C$ in NLG_0^{**}.

Case 3: The last step in the proof is an R2 rule with $X \in BSTp$, but $X \notin Tp$.

$$\frac{\begin{array}{cc}\vdots & \vdots \\ X \to y & Y[x] \to z\end{array}}{Y[(x/y, X)] \to z}\ [\mathrm{R2}']$$

The induction hypothesis says that $X \to y$ has a proof in NLG_0^{**}. The last rule applied in the proof of this sequent cannot be an R1 rule, because $X \notin Tp$. So the last rule in the proof of $X \to y$ is an R2 rule, say R2′ (the R2″ case is similar). We have the following proof:

$$\frac{\dfrac{\begin{array}{cc}\vdots & \vdots \\ W \to v & Z[d] \to y\end{array}}{Z[(d/v, W)] \to y}\ [\mathrm{R2}'] \qquad \begin{array}{c}\vdots \\ Y[x] \to z\end{array}}{Y[(x/y, Z[(d/v, W)])] \to z}\ [\mathrm{R2}']$$

where W is in Tp. We can transform it as follows:

$$\dfrac{\begin{matrix}\vdots\\ W \to v\end{matrix} \qquad \dfrac{\begin{matrix}\vdots\\ Z[d] \to y\end{matrix} \quad \begin{matrix}\vdots\\ Y[x] \to z\end{matrix}}{Y[(x/y, Z[d])] \to z}\,[\text{R2}']}{Y[(x/y, Z[(d/v, W)])] \to z}\,[\text{R2}']$$

The sequents $W \to v$ and $Y[(x/y, Z[d])] \to z$ have proofs in NLG_0^{**} (induction hypothesis) and therefore $Y[(x/y, Z[(d/v, W)])] \to z$ has a proof in NLG_0^{**} too. End of proof of Theorem 2.1.2. □

2.1.3. THEOREM. *Anything derivable in NLG_0^{**} is also derivable in NLG_0.*

Proof: This is trivial, any proof in NLG_0^{**} is also a proof in NLG_0. □

From Theorems 2.1.1, 2.1.2 and 2.1.3 we know that $\text{NLG}_0 = \text{NLG}_0^{*} = \text{NLG}_0^{**}$. We introduce another calculus now, called A1–R2–R3–R4. This calculus is defined as follows:

$$[\text{A1}]\ \ x \to x\ , \text{for } x \in Tp$$

$$\dfrac{w \to y \quad z \to x}{w \to x/(y\backslash z)}\,[\text{R3'}] \qquad \dfrac{w \to y \quad z \to x}{w \to (z/y)\backslash x}\,[\text{R3''}]$$

$$\dfrac{w \to y \quad z \to x}{w/x \to y/z}\,[\text{R4'}] \qquad \dfrac{w \to y \quad z \to x}{x\backslash w \to z\backslash y}\,[\text{R4''}]$$

$$\dfrac{w \to y \quad Y[x] \to z}{Y[(x/y, w)] \to z}\,[\text{R2}'] \qquad \dfrac{w \to y \quad Y[x] \to z}{Y[(w, y\backslash x)] \to z}\,[\text{R2}'']$$

for $Y \in BSTp$ and $w, x, y, z \in Tp$.

Figure 2.3: A1–R2–R3–R4

2.1.4. THEOREM. *A1–R2–R3–R4 = NLG_0^{**}*

Proof: Let us repeat the calculus NLG_0^{**} here.

The last step in the proof of the premise of an R1 rule must be an R2 rule. Therefore we can replace R1′ and R1″ by the following rules:

$$\dfrac{w \to y \quad z \to x}{w \to x/(y\backslash z)}\,[\text{R3'} = \text{R2}'' + \text{R1}'] \qquad \dfrac{w \to y \quad z \to x}{w \to (z/y)\backslash x}\,[\text{R3''} = \text{R2}' + \text{R1}'']$$

$$\dfrac{w \to y \quad z \to x}{w/x \to y/z}\,[\text{R4'} = \text{R2}' + \text{R1}'] \qquad \dfrac{w \to y \quad z \to x}{x\backslash w \to z\backslash y}\,[\text{R4''} = \text{R2}'' + \text{R1}'']$$

This proves Theorem 2.1.4. □

Observe that the antecedent of the conclusion of an R2 rule is not in Tp. On the other hand, the antecedents of the premises and the conclusions of R3 and R4 rules

$$\text{[A1]}\quad x \to x \text{ , for } x \in Tp$$

$$\frac{(w,y) \to x}{w \to x/y}\ \text{[R1$'$]} \qquad \frac{(y,w) \to x}{w \to y\backslash x}\ \text{[R1$''$]}$$

$$\frac{w \to y \quad Y[x] \to z}{Y[(x/y,w)] \to z}\ \text{[R2$'$]} \qquad \frac{w \to y \quad Y[x] \to z}{Y[(w,y\backslash x)] \to z}\ \text{[R2$''$]}$$

for $Y \in BSTp$ and $w,x,y,z \in Tp$.

Figure 2.4: NLG_0^{**}

are in Tp. Hence, a proof in A1–R2–R3–R4 of a sequent with an antecedent in Tp contains R3 and R4 rules only.

2.2 Recognition for Categorial Grammars Based on NL

We have come to a point now where we can remove the brackets from our calculi. We define the bracket-free calculus NLG_1. The rules of NLG_1 are:

$$\text{[A1]}\quad x \to x \text{ , for } x \in Tp$$

$$\frac{w \to y \quad z \to x}{w \to x/(y\backslash z)}\ \text{[R3']} \qquad \frac{w \to y \quad z \to x}{w \to (z/y)\backslash x}\ \text{[R3"]}$$

$$\frac{w \to y \quad z \to x}{w/x \to y/z}\ \text{[R4']} \qquad \frac{w \to y \quad z \to x}{x\backslash w \to z\backslash y}\ \text{[R4"]}$$

$$\frac{x \to y \quad \Delta, w, \Delta' \to z}{\Delta, w/y, x, \Delta' \to z}\ \text{[R5$'$]} \qquad \frac{x \to y \quad \Delta, w, \Delta' \to z}{\Delta, x, y\backslash w, \Delta' \to z}\ \text{[R5$''$]}$$

for $\Delta, \Delta' \in STp$ and $w,x,y,z \in Tp$.

Figure 2.5: NLG_1

This calculus differs from A1–R2–R3–R4 because the brackets in the rules R2′ and R2″ are erased. This results in two new rules R5′ and R5″.

There is a one-to-one correspondence between proofs in A1–R2–R3–R4 and NLG_1. We can prove the following lemma's:

2.2.1. LEMMA. *If a bracketed sequent $X \to z$ is derivable in A1–R2–R3–R4, then $yield(X) \to z$ is derivable in NLG_1.*

Proof: A proof in A1–R2–R3–R4 has the following shape:

$$\dfrac{\dfrac{x_1 \overset{\vdots}{\to} w_1 \quad \dfrac{x_2 \overset{\vdots}{\to} w_2 \quad \dfrac{x_3 \overset{\vdots}{\to} w_3 \quad Y_3 \overset{\vdots}{\to} z}{Y_2 \to z}\,[\text{R2}]}{Y_1 \to z}\,[\text{R2}]}{Y_0 \to z}\,[\text{R2}]$$

(each $\overset{\vdots}{\to}$ stands for R3-R4)

The $Y_i \to z$ premises are the only premises containing brackets. When we leave out those brackets, we immediately obtain an NLG_1 proof. □

2.2.2. LEMMA. *If a sequent $Y \to z$ is derivable in NLG_1, then there is an X such that $yield(X) = Y$ and $X \to z$ is derivable in A1–R2–R3–R4.*

Proof: If we have an NLG_1 proof it has the following shape:

$$\dfrac{\dfrac{x_1 \overset{\vdots}{\to} w_1 \quad \dfrac{x_2 \overset{\vdots}{\to} w_2 \quad \dfrac{x_3 \overset{\vdots}{\to} w_3 \quad Y_3 \overset{\vdots}{\to} z}{Y_2 \to z}\,[\text{R5}]}{Y_1 \to z}\,[\text{R5}]}{Y_0 \to z}\,[\text{R5}]$$

(each $\overset{\vdots}{\to}$ stands for R3-R4)

It is easy to add brackets to the Y_i such that we obtain an A1–R2–R3–R4 proof. □

Lemma's 2.2.1 and 2.2.2 enable us to remove the brackets in the definition of grammaticality as well.

$L(G)$ has been defined as the set of strings $w \in \Sigma^*$, $w = a_1 \dots a_n$ such that for some *bracketed* string of types X, $yield(X) = A_1, \dots, A_n$, $\langle a_i, A_i \rangle \in Lex$ $(1 \leq i \leq n)$ and $X \to s$ is derivable in NLG_0.

But the language can be defined bracket-free now: $L'(G)$ is the set of strings $w \in \Sigma^*$, $w = a_1 \dots a_n$ such that for some string of types $A_1, \dots, A_n$, $\langle a_i, A_i \rangle \in Lex$ $(1 \leq i \leq n)$ and $A_1, \dots, A_n \to s$ in NLG_1.

2.2.3. LEMMA. *For every lexicon G, $L'(G) = L(G)$.*

Proof: Follows from Lemma's 2.2.1 and 2.2.2, Theorem 2.1.4 and the fact that $\text{NLG}_0 = \text{NLG}_0^{**}$. □

2.2.4. LEMMA. *For all $x, y \in Tp$, we can check in time $\mathcal{O}((|x| + |y|)^2)$ whether $x \to y$ is provable in NLG_1.*

Proof: The antecedents of the premises and of the conclusions of the R3 and R4 rules have length one. The antecedent of the conclusion of an R5 rule has length bigger than one. Therefore, the proof of $x \to y$ consists of R3 and R4 rules only. The subformula property holds for the "R3–R4 calculus". Let n be the number of slashes in x plus the number of slashes in y. n is linear in $|x| + |y|$. We have to compute at most $\mathcal{O}(n^2)$ times whether some $a \to b$ is derivable (because a and b are subformulas of x and y). We *memoize* (Cormen, Leiserson and Rivest 1990, pp. 312–314) the results of the attempts to prove something: the first time we have to compute $a \to b$ the result

of the computation is stored in a table. If we have to compute it again later we look up the answer in the table. The search space is a graph with nodes labeled $x \to y$. As said, there are $\mathcal{O}(n^2)$ nodes. Every node has at most four outgoing arcs: at most two rules of R3–R4 are applicable, so we have to prove at most 4 premises. Therefore, the number of arcs is $\mathcal{O}(4n^2) = \mathcal{O}(n^2)$. The algorithm is a depth first traversal of the graph. Because of memoization, every arc is traversed only once. The algorithm takes time $\mathcal{O}(n^2)$ at most. □

2.2.5. THEOREM. *The recognition problem for NL based categorial grammar can be reduced to context-free grammar recognition in polynomial time.*

Proof: We take the new definition of categorial languages: $L'(G)$ is the set of strings $w \in \Sigma^*$, $w = a_1 \dots a_n$ such that for some string of types $A_1, \dots, A_n$, $\langle a_i, A_i \rangle \in Lex$ $(1 \leq i \leq n)$ and $A_1, \dots, A_n \to s$ in NLG_1. The symbol s is the distinguished type of the grammar ($s \in Pr$). Proofs in NLG_1 look like:

$$\dfrac{\dfrac{\dfrac{\vdots\ R3\text{-}R4}{x_1 \to w_1} \quad \dfrac{\dfrac{\vdots\ R3\text{-}R4}{x_2 \to w_2} \quad \dfrac{\dfrac{\vdots\ R3\text{-}R4}{x_3 \to w_3} \quad \dfrac{x_n \to w_n \quad s \to s}{\vdots\ Y_3 \to s}\,[\mathrm{R5}]}{Y_2 \to s}\,[\mathrm{R5}]}{Y_1 \to s}\,[\mathrm{R5}]}{Y_0 \to s}$$

The sequence $s, \dots, Y_3, Y_2, Y_1, Y_0$ can be seen as a derivation in a context-free grammar. Given a lexicon, we construct a context-free grammar that generates the same language as the categorial grammar. The context-free grammar consists of binary and unary grammar rules. The start symbol is s. The binary rules simulate rewriting the symbols $s, \dots, Y_3, Y_2, Y_1, Y_0$. The unary rules simulate the lexical type assignment.

The binary rules are of the form $w \Rightarrow w/y, x$ and $w \Rightarrow x, y\backslash w$. Let the variables x, w/y, and $y\backslash w$ range over all possible subtypes in the lexicon. The rule $w \Rightarrow w/y, x$ (or $w \Rightarrow x, y\backslash w$) is added to the grammar when the sequent $x \to y$ is derivable in NLG_1. The unary rules (for lexical type assignment) are of the form $A \Rightarrow a$ with $\langle a, A \rangle \in Lex$.

The number of subtypes in the lexicon is linear in the size of the lexicon. The number of binary rules is quadratic in the number of subtypes. Therefore, the size of the grammar is quadratic in the size of the lexicon. Construction of the grammar takes polynomial time because we can compute $x \to y$ in polynomial time. □

The time complexity of context-free grammar recognition is $\mathcal{O}(|G|n^3)$ where $|G|$ is the size of the grammar and n the length of the input sentence (Sippu and Soisalon-Soininen 1988, p. 147). Via construction of the context-free grammar, we have a polynomial time algorithm for recognition in the NL based categorial grammar. After construction of the grammar, the time complexity of recognition is cubic in the length of the string and quadratic in the size of the lexicon.

A polynomial time algorithm for deciding provability in NLG_1 can be given too. Because we do not have a lexicon anymore, the number of possible types is infinite.

We cannot use a context-free grammar in the style described here because it would have infinitely many rules. But instead of computing the grammar in advance we can compute grammar rules "on the fly". We try to combine adjacent types according to the schemes $w \Rightarrow w/y, x$ and $w \Rightarrow x, y\backslash w$ and compute whether $x \to y$ is derivable in R3–R4.

Later in this thesis (section 7.1) we will give an alternative proof of the main theorem. We will give an implementation in Prolog of an algorithm that recognizes sentences of the NL based categorial grammar. We prove that this implementation needs polynomial time to decide on grammaticality.

Chapter 3

The Second Order Fragment of L

In this chapter[1] we show that when we restrict ourselves to the *second order fragment* of L, we can give a recognition algorithm that runs in polynomial time. The structure of the section is as follows. First, we define the second order fragment of L. This fragment is a fragment with limited depth of nesting of the types. It has nothing to do with second order logic or polymorphism. These are extensions, not fragments, of the basic calculus. We prove that the second order fragment is equivalent with another calculus called ApplComp. ApplComp stands for Application and Composition. In order to show the equivalence we introduce a new notation for types, and an auxiliary calculus called Aux. Once we have the calculus ApplComp we can give a polynomial time recognition algorithm. This will be done in the last section.

3.1 Categorial Grammars

For convenience we repeat the definitions given earlier. Because we are in an associative fragment now we can simplify things a little. Instead of bracketed strings of types we use ordinary strings now. A Categorial grammar G is defined by a lexicon Lex and a calculus C. A *lexicon* Lex is a finite relation between Σ and Tp ($Lex \subset \Sigma \times Tp$). Tp is the set of *types*. A (finite) set of *primitive types* Pr is given. Types are constructed from primitive types by two type-forming operators: / and \, i.e., Tp is the smallest set containing Pr such that if $A, B \in Tp$, then $A\backslash B, B/A \in Tp$. One member s of Pr is singled out as the *distinguished type*. If a lexicon Lex relates $a \in \Sigma$ with $A \in Tp$ ($\langle a, A\rangle \in Lex$), we say Lex *assigns* A *to* a.

Besides the lexicon, we need some calculus C. C consists of axioms and rules. We say that a finite sequence of types $A_1, \ldots, A_n$ *cancels* to a type A_{n+1} if the expression $A_1, \ldots, A_n \rightarrow A_{n+1}$ is derivable in the calculus. Now L(G) is defined to be the set of strings $s = a_1 \ldots a_n$ such that for some types $A_1, \ldots, A_n, \langle a_i, A_i\rangle \in Lex (1 \leq i \leq n)$ and $A_1, \ldots, A_n$ cancels to s in C. The calculus we take as our starting point is L:

[1] This chapter is based on Aarts (1994b) and Aarts (1994a).

$$\mathsf{A} \to \mathsf{A}$$

Left rules

$$\frac{\Gamma \to \mathsf{A} \quad \Delta, \mathsf{B}, \Delta' \to \mathsf{C}}{\Delta, \mathsf{B/A}, \Gamma, \Delta' \to \mathsf{C}}\ [/\mathrm{L}] \qquad \frac{\Gamma, \mathsf{A} \to \mathsf{B}}{\Gamma \to \mathsf{B/A}}\ [/\mathrm{R}]$$

$$\frac{\Gamma \to \mathsf{A} \quad \Delta, \mathsf{B}, \Delta' \to \mathsf{C}}{\Delta, \Gamma, \mathsf{A\backslash B}, \Delta' \to \mathsf{C}}\ [\backslash \mathrm{L}] \qquad \frac{\mathsf{A}, \Gamma \to \mathsf{B}}{\Gamma \to \mathsf{A\backslash B}}\ [\backslash \mathrm{R}]$$

Right rules

Figure 3.1: L

(Γ and Δ stand for finite sequences of types). An example lexicon *Lex* is:

$$\{\langle everyone, pn\rangle, \langle everyone, s/(np\backslash s)\rangle, \langle loves, (np\backslash s)/np\rangle, \langle somebody, pn\rangle, \langle somebody, (s/np)\backslash s\rangle\}$$

We can write this lexicon also as:

everyone	pn
	$s/(np\backslash s)$
loves	$(np\backslash s)/np$
somebody	pn
	$(s/np)\backslash s$

The string "everyone loves somebody" is grammatical: $\langle everyone, s/(np\backslash s)\rangle, \langle loves, (np\backslash s)/np\rangle, \langle somebody, (s/np)\backslash s\rangle$ are in *Lex*, and the sequent "$s/(np\backslash s), (np\backslash s)/np (s/np)\backslash s \to s$" is derivable in L:

$$\dfrac{\dfrac{np \to np \quad \dfrac{np \to np \quad s \to s}{np, np\backslash s \to s}\,[\backslash \mathrm{L}]}{np, (np\backslash s)/np, np \to s}\,[/\mathrm{L}]}{\dfrac{np, (np\backslash s)/np \to s/np}{np, (np\backslash s)/np, (s/np)\backslash s \to s}\,[/\mathrm{R}] \quad s \to s}\,[\backslash \mathrm{L}]}{\dfrac{(np\backslash s)/np, (s/np)\backslash s \to np\backslash s}{s/(np\backslash s), (np\backslash s)/np, (s/np)\backslash s \to s}\,[\backslash \mathrm{R}] \quad s \to s}\,[/\mathrm{L}]$$

We define the *order of a type*:

$$\begin{aligned} order(A) &= 0 \text{ if } A \text{ is a primitive type} \\ order(A/B) &= max(order(A), order(B)+1) \\ order(B\backslash A) &= max(order(A), order(B)+1) \end{aligned}$$

The *order of a sequent* is defined as the order of the highest order type occurring in

it. We restrict ourselves to grammars in which words have types of order at most 2. This is called the second order fragment. We will write it as L2.

3.2 The System Aux

We introduce a new notation that makes types flatter. Instead of nesting the arguments with slashes we introduce *lists of arguments*. Slash types can be translated into list types. An example is: $a\backslash(b\backslash((c/d)/e)) \rightsquigarrow (a, b \Rightarrow c \Leftarrow d, e)$. The result type in the middle must be primitive. This notation can also be found in Buszkowski (1990) although the order in the argument lists here is the reverse of the order in Buszkowski's notation. An example of a derivable sequent is:

$$b, a, (a, b \Rightarrow c \Leftarrow d, e), e, d \rightarrow c.$$

The example lexicon in the new notation:

everyone	pn
	$(s \Leftarrow (np \Rightarrow s))$
loves	$(np \Rightarrow s \Leftarrow np)$
somebody	pn
	$((s \Leftarrow np) \Rightarrow s)$

We can define L2 in this new notation as follows (($[\ ] \Rightarrow a \Leftarrow [\ ]$) equals a):

$\mathsf{A} \rightarrow \mathsf{A}$ (A *primitive*)

$$\frac{\Gamma \rightarrow \mathsf{A} \quad \Delta, (\mathsf{L} \Rightarrow \mathsf{B} \Leftarrow \mathsf{T}), \Delta' \rightarrow \mathsf{C}}{\Delta, (\mathsf{L} \Rightarrow \mathsf{B} \Leftarrow \mathsf{T}, \mathsf{A}), \Gamma, \Delta' \rightarrow \mathsf{C}}\ [/\mathrm{L}] \qquad \frac{\Gamma, \mathsf{A} \rightarrow (\mathsf{L} \Rightarrow \mathsf{B} \Leftarrow \mathsf{T})}{\Gamma \rightarrow (\mathsf{L} \Rightarrow \mathsf{B} \Leftarrow \mathsf{T}, \mathsf{A})}\ [/\mathrm{R}]$$

$$\frac{\Gamma \rightarrow \mathsf{A} \quad \Delta, (\mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{L}), \Delta' \rightarrow \mathsf{C}}{\Delta, \Gamma, (\mathsf{A}, \mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{L}), \Delta' \rightarrow \mathsf{C}}\ [\backslash\mathrm{L}] \qquad \frac{\mathsf{A}, \Gamma \rightarrow (\mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{L})}{\Gamma \rightarrow (\mathsf{A}, \mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{L})}\ [\backslash\mathrm{R}]$$

A and B types, Γ, Δ, Δ', L and T (possibly empty) lists of types.

Figure 3.2: L2 in sub-categorization list notation

The system is equivalent to L because L is associative. The auxiliary system *Aux* can be defined as follows. The left rules [\L], [/L] and the axioms remain as they are. The right rules of *Aux* are:

$$\frac{\Gamma, (\mathsf{V} \Rightarrow \mathsf{C} \Leftarrow \mathsf{W}) \rightarrow (\mathsf{L} \Rightarrow \mathsf{B} \Leftarrow \mathsf{T})}{\Gamma, (\mathsf{V} \Rightarrow \mathsf{C} \Leftarrow \mathsf{W}, \mathsf{A}) \rightarrow (\mathsf{L} \Rightarrow \mathsf{B} \Leftarrow \mathsf{T}, \mathsf{A})}\ [/\mathrm{R}_*]$$

$$\frac{(\mathsf{V} \Rightarrow \mathsf{C} \Leftarrow \mathsf{W}), \Gamma \rightarrow (\mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{L})}{(\mathsf{A}, \mathsf{V} \Rightarrow \mathsf{C} \Leftarrow \mathsf{W}), \Gamma \rightarrow (\mathsf{A}, \mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{L})}\ [\backslash\mathrm{R}_*]$$

Again, A and B types, Γ, V, W, L and T lists of types.

3.2.1. THEOREM. *L2 = Aux, i.e. a sequent is derivable in L2 iff it is derivable in Aux.*

Before we prove this theorem we prove a lemma:

3.2.2. LEMMA. *If* $\Gamma, B \to E$ *with* B *atomic is provable in Aux then*

- Γ *is empty and* $E = B$ or
- *There are* $\mathsf{C}_1, \ldots, \mathsf{C}_m$ *and* $\Theta_1, \ldots, \Theta_m$ *such that* Γ *is of the form* $\Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'', \mathsf{B}, \mathsf{C}_m, \ldots, \mathsf{C}_1), \Theta_1, \ldots, \Theta_m$ *and the following sequents are provable:*
  - $\Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'') \to \mathsf{E}$
  - *for all* i, $1 \leq i \leq m$, $\Theta_i \to \mathsf{C}_i$

Proof: We prove this with strong induction on the size of a sequent (where we can define the size as the number of characters needed to write down the sequent). Lemma 3.2.2 obviously holds for axioms.

There are three possibilities for the last rule used in the proof of $\Gamma, B \to E$. It can be [\L], [/L] or [$\backslash \mathrm{R}_*$].

- [\L] This is easy. B is in the right premise. Lemma 3.2.2 holds for the right premise, and therefore is of the form:
  $\Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'', \mathsf{B}, \mathsf{C}_m, \ldots, \mathsf{C}_1), \Theta_1, \ldots, \Theta_m, \mathsf{B} \to \mathsf{E}$
  Suppose the left premise is of the form $\Xi \to \mathsf{A}$. Then the conclusion $\Gamma, B \to E$ is of the form:
  - $\Delta, \Xi, (\mathsf{A}, \Delta' \Rightarrow D \Leftarrow \Delta'', \mathsf{B}, \mathsf{C}_m, \ldots, \mathsf{C}_1), \Theta_1, \ldots, \Theta_m, \mathsf{B} \to \mathsf{E}$
  - $\Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'', \mathsf{B}, \mathsf{C}_m, \ldots, \mathsf{C}_1, \mathsf{A}), \Xi, \Theta_1, \ldots, \Theta_m, \mathsf{B} \to \mathsf{E}$
  - $\Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'', \mathsf{B}, \mathsf{C}_m, \ldots, \mathsf{C}_1), \Theta_1, \ldots, \Theta_{i-1}, \Theta'_i, \Theta_{i+1}, \ldots \Theta_m, \mathsf{B} \to \mathsf{E}$
  - $\Delta''', (\Delta' \Rightarrow D \Leftarrow \Delta'', \mathsf{B}, \mathsf{C}_m, \ldots, \mathsf{C}_1), \Theta_1, \ldots, \Theta_m, \mathsf{B} \to \mathsf{E}$

  In all three cases the lemma holds for the conclusion.
- [/L] B can be in the right premise or in the left premise.
  - B in the right premise: Lemma 3.2.2 holds for the right premise, and therefore is of the form:
    $\Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'', \mathsf{B}, \mathsf{C}_m, \ldots, \mathsf{C}_1), \Theta_1, \ldots, \Theta_m, \mathsf{B} \to \mathsf{E}$
    Suppose the left premise is of the form $\Xi \to \mathsf{A}$. Then the conclusion $\Gamma, B \to E$ is of the form:
    - $\Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'', \mathsf{B}, \mathsf{C}_m, \ldots, \mathsf{C}_2, \mathsf{A}), \Xi, \Theta_2, \ldots, \Theta_m, \mathsf{B} \to \mathsf{E}$
    - $\Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'', \mathsf{B}, \mathsf{C}_m, \ldots, \mathsf{C}_1), \Theta_1, \ldots, \Theta_{i-1}, \Theta'_i, \Theta_{i+1}, \ldots \Theta_m, \mathsf{B} \to \mathsf{E}$
    - $\Delta''', (\Delta' \Rightarrow D \Leftarrow \Delta'', \mathsf{B}, \mathsf{C}_m, \ldots, \mathsf{C}_1), \Theta_1, \ldots, \Theta_m, \mathsf{B} \to \mathsf{E}$

    In all four cases the lemma holds for the conclusion.
  - B in the left premise. We have the following situation:

$$\frac{\Theta, \mathsf{B} \to \mathsf{A} \quad \Xi, (\Xi' \Rightarrow D \Leftarrow \Xi'') \to \mathsf{F}}{\Xi, (\Xi' \Rightarrow D \Leftarrow \Xi'', \mathsf{A}), \Theta, \mathsf{B} \to \mathsf{F}} \; /L$$

    The lemma holds for $\Theta, \mathsf{B} \to \mathsf{A}$. We know that the last step in the proof is:

$$\frac{\Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'', \mathsf{B}, \mathsf{C}_m, \ldots, \mathsf{C}_1), \Theta_1, \ldots, \Theta_m, \mathsf{B} \to \mathsf{A} \quad \Xi, (\Xi' \Rightarrow D \Leftarrow \Xi'') \to \mathsf{F}}{\Xi, (\Xi' \Rightarrow D \Leftarrow \Xi'', \mathsf{A}), \Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'', \mathsf{B}, \mathsf{C}_m, \ldots, \mathsf{C}_1), \Theta_1, \ldots, \Theta_m, \mathsf{B} \to \mathsf{F}} \; [/\mathrm{L}]$$

The lemma holds for the conclusion.

- $[\backslash R_*]$

$$\frac{(V \Rightarrow C \Leftarrow W), \Gamma, B \to (T \Rightarrow D \Leftarrow L)}{(A, V \Rightarrow C \Leftarrow W), \Gamma, B \to (A, T \Rightarrow D \Leftarrow L)}\ [\backslash R_*]$$

The lemma holds for the premise, therefore the last step is of the form:

$$\frac{(V \Rightarrow C \Leftarrow W), \Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'', B, C_m, \ldots, C_1), \Theta_1, \ldots, \Theta_m, B \to (T \Rightarrow D \Leftarrow L)}{(A, V \Rightarrow C \Leftarrow W), \Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'', B, C_m, \ldots, C_1), \Theta_1, \ldots, \Theta_m, B \to (A, T \Rightarrow D \Leftarrow L)}\ [\backslash R_*]$$

or

$$\frac{(V \Rightarrow C \Leftarrow \Delta'', B, C_m, \ldots, C_1), \Theta_1, \ldots, \Theta_m, B \to (T \Rightarrow D \Leftarrow L)}{(A, V \Rightarrow C \Leftarrow \Delta'', B, C_m, \ldots, C_1), \Theta_1, \ldots, \Theta_m, B \to (A, T \Rightarrow D \Leftarrow L)}\ [\backslash R_*]$$

In both cases, the lemma holds for the conclusion.

End of proof of Lemma 3.2.2.

Proof of theorem 3.2.1 This proof consists of two parts: $Aux \subseteq L2$ and $L2 \subseteq Aux$. It is easy to see that $Aux \subseteq L2$. In any Aux proof we can replace applications of $[\backslash R_*]$ by:

$$\frac{\dfrac{(V \Rightarrow C \Leftarrow W), \Gamma \to (T \Rightarrow B \Leftarrow L)}{A, (A, V \Rightarrow C \Leftarrow W), \Gamma \to (T \Rightarrow B \Leftarrow L)}\ [\backslash L]}{(A, V \Rightarrow C \Leftarrow W), \Gamma \to (A, T \Rightarrow B \Leftarrow L)}\ [\backslash R]$$

and we obtain a L2 proof.

The proof of $L2 \subseteq Aux$.

$L2$ contains the following rule:

$$\frac{\Gamma, B \vdash (L \Rightarrow A \Leftarrow R)}{\Gamma \vdash (L \Rightarrow A \Leftarrow R, B)}\ [/R]$$

We have to prove now that if we have an Aux-proof of the premise, we also have an Aux-proof of the conclusion.

We know B is atomic (we are in the second order fragment). Lemma 3.2.2 holds. Therefore either $\Gamma, B \to (L \Rightarrow A \Leftarrow R)$ is an axiom or it equals
$\Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'', B, C_m, \ldots, C_1), \Theta_1, \ldots, \Theta_m, B \to (L \Rightarrow A \Leftarrow R)$

In the second case, apply $[/L_*]$ m times on the $\Theta_i \to C_i$. This gives you the conclusion.

$$\frac{\Theta_1 \to C_1 \qquad \dfrac{\Theta_2 \to C_2 \quad \dfrac{\Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'') \to (L \Rightarrow A \Leftarrow R)}{\Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'', B) \to (L \Rightarrow A \Leftarrow R, B)}\ [/R_*] \quad \vdots \quad \Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'', B, C_m, \ldots, C_3), \Theta_3, \ldots, \Theta_m \to (L \Rightarrow A \Leftarrow R, B)}{\Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'', B, C_m, \ldots, C_2), \Theta_2, \ldots, \Theta_m \to (L \Rightarrow A \Leftarrow R, B)}\ [/L_*]}{\Delta, (\Delta' \Rightarrow D \Leftarrow \Delta'', B, C_m, \ldots, C_1), \Theta_1, \ldots, \Theta_m \to (L \Rightarrow A \Leftarrow R, B)}\ [/L_*]$$

For $[\backslash R]$ the proof is similar. We can prove the mirror image of Lemma 3.2.2 and use this to prove $L2 \subseteq Aux$ in case the last step in the L2 proof is $[\backslash R]$.

End of proof of Theorem 3.2.1.

3.3 Cut Elimination in Aux

In this section we prove that the Cut rule is an admissible rule in Aux, i.e., Aux with Cut is equivalent with Cut. Aux has been defined as follows:

Axioms

$$\mathsf{A} \to \mathsf{A} \quad (\mathsf{A}\ \textit{primitive})$$

$$\frac{\Gamma \to \mathsf{A} \quad \Delta, (\mathsf{L} \Rightarrow \mathsf{B} \Leftarrow \mathsf{T}), \Delta' \to \mathsf{C}}{\Delta, (\mathsf{L} \Rightarrow \mathsf{B} \Leftarrow \mathsf{T}, \mathsf{A}), \Gamma, \Delta' \to \mathsf{C}}\ [/\mathrm{L}] \qquad \frac{\Gamma, (\mathsf{V} \Rightarrow \mathsf{C} \Leftarrow \mathsf{W}) \to (\mathsf{L} \Rightarrow \mathsf{B} \Leftarrow \mathsf{T})}{\Gamma, (\mathsf{V} \Rightarrow \mathsf{C} \Leftarrow \mathsf{W}, \mathsf{A}) \to (\mathsf{L} \Rightarrow \mathsf{B} \Leftarrow \mathsf{T}, \mathsf{A})}\ [/\mathrm{R}_*]$$

$$\frac{\Gamma \to \mathsf{A} \quad \Delta, (\mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{L}), \Delta' \to \mathsf{C}}{\Delta, \Gamma, (\mathsf{A}, \mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{L}), \Delta' \to \mathsf{C}}\ [\backslash \mathrm{L}] \qquad \frac{(\mathsf{V} \Rightarrow \mathsf{C} \Leftarrow \mathsf{W}), \Gamma \to (\mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{L})}{(\mathsf{A}, \mathsf{V} \Rightarrow \mathsf{C} \Leftarrow \mathsf{W}), \Gamma \to (\mathsf{A}, \mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{L})}\ [\backslash \mathrm{R}_*]$$

Figure 3.3: The system Aux

The types $(\mathsf{A}, \mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{L})$ and $(\mathsf{L} \Rightarrow \mathsf{B} \Leftarrow \mathsf{T}, \mathsf{A})$ in the conclusions of the inference rules are called the *principal types*.

The Cut rule is:

$$\frac{\Gamma \to \mathsf{A} \quad \Delta, \mathsf{A}, \Delta' \to \mathsf{B}}{\Delta, \Gamma, \Delta' \to \mathsf{B}}\ [\mathrm{Cut}]$$

3.3.1. THEOREM. *Aux* + [Cut] = *Aux*.

Proof: The proof is based on the standard Cut elimination proof for Lambek calculus, e.g., Hendriks (1993, p. 194 ff.).

The structure of the cut elimination proof is as follows. Cut rules in proofs are "pushed upwards". We keep pushing the Cut rule upward until one of the premises of the Cut is an axiom. Then the Cut can be removed. First, a *degree* of a Cut is defined. In order to push the cut upwards we consider the last step in the proofs of both premises of the Cut rule. Depending on these last steps we transform the proof. The Cut rule is replaced by another Cut rule of a lower degree. Repeating this, we will get a proof in which one of the premises of the Cut rule is an axiom.

The degree of a sequent $d(\Gamma \to \mathsf{A})$ is defined as the number of connectives in the types. The degree $d(\alpha)$ of a Cut inference α

$$\frac{\Gamma \to \mathsf{A} \quad \Delta, \mathsf{A}, \Delta' \to \mathsf{B}}{\Delta, \Gamma, \Delta' \to \mathsf{B}}\ [\mathrm{Cut}]$$

equals $d(\Gamma) + d(\Delta) + d(\Delta') + d(\mathsf{A}) + d(\mathsf{B})$.

If the last rule applied in the proof of $\Gamma \to \mathsf{A}$ is a left rule we can simply permute the left rule and the Cut rule. If the last rule in the proof of $\Delta, \mathsf{A}, \Delta' \to \mathsf{B}$ is a left rule and A is not the principal type in the left rule we can also permute the left rule and the Cut rule. An example of this is:

$$\dfrac{\Gamma \to A \quad \dfrac{\Delta \to D \quad (\ldots \Rightarrow C \Leftarrow \ldots), \Delta', A, \Delta'' \to C}{\Delta, (D, \ldots \Rightarrow C \Leftarrow \ldots), \Delta', A, \Delta'' \to C}\,[\backslash L]}{\Delta, (D, \ldots \Rightarrow C \Leftarrow \ldots), \Delta', \Gamma, \Delta'' \to C}\,[\text{Cut}]$$

is transformed into

$$\dfrac{\Delta \to D \quad \dfrac{\Gamma \to A \quad (\ldots \Rightarrow C \Leftarrow \ldots), \Delta', A, \Delta'' \to C}{(\ldots \Rightarrow C \Leftarrow \ldots), \Delta', \Gamma, \Delta'' \to C}\,[\text{Cut}]}{\Delta, (D, \ldots \Rightarrow C \Leftarrow \ldots), \Delta', \Gamma, \Delta'' \to C}\,[\backslash L]$$

Two hard cases are left:

- the last step in the proof of $\Gamma \to A$ is a right rule and the last step in the proof of $\Delta, A, \Delta' \to B$ is a left rule and the cut formula A is the principal type (case 1)
- the last step in the proof of $\Gamma \to A$ is a right rule and the last step in the proof of $\Delta, A, \Delta' \to B$ is a right rule (case 2)

Case 1a: The R_* rule and the L rule have the same direction.

$$\dfrac{\dfrac{\Gamma, (S \Rightarrow D \Leftarrow U) \to (L' \Rightarrow B' \Leftarrow T')}{\Gamma, (S \Rightarrow D \Leftarrow U, A) \to (L' \Rightarrow B' \Leftarrow T', A)}\,[/R_*] \quad \dfrac{\Delta'' \to A \quad \Delta, (L' \Rightarrow B' \Leftarrow T'), \Delta' \to B}{\Delta, (L' \Rightarrow B' \Leftarrow T', A), \Delta'', \Delta' \to B}\,[/L]}{\Delta, \Gamma, (S \Rightarrow D \Leftarrow U, A), \Delta'', \Delta' \to B}\,[\text{Cut}]$$

transform this into:

$$\dfrac{\Delta'' \to A \quad \dfrac{\Gamma, (S \Rightarrow D \Leftarrow U) \to (L' \Rightarrow B' \Leftarrow T') \quad \Delta, (L' \Rightarrow B' \Leftarrow T'), \Delta' \to B}{\Delta, \Gamma, (S \Rightarrow D \Leftarrow U), \Delta' \to B}\,[\text{Cut}]}{\Delta, \Gamma, (S \Rightarrow D \Leftarrow U, A), \Delta'', \Delta' \to B}\,[/L]$$

Case 1b: The R_* rule and the L rule do not have the same direction.

$$\dfrac{\dfrac{\Gamma, (S \Rightarrow D \Leftarrow U) \to (C, L' \Rightarrow B' \Leftarrow T')}{\Gamma, (S \Rightarrow D \Leftarrow U, A) \to (C, L' \Rightarrow B' \Leftarrow T', A)}\,[/R_*] \quad \dfrac{\Delta'' \to C \quad \Delta, (L' \Rightarrow B' \Leftarrow T', A), \Delta' \to B}{\Delta, \Delta'', (C, L' \Rightarrow B' \Leftarrow T', A), \Delta' \to B}\,[\backslash L]}{\Delta, \Delta'', \Gamma, (S \Rightarrow D \Leftarrow U, A), \Delta' \to B}\,[\text{Cut}]$$

We rearrange the steps in the proof of $\Gamma, (S \Rightarrow D \Leftarrow U, A) \to (C, L' \Rightarrow B' \Leftarrow T', A)$ such that the last step is either a left rule or a $[\backslash R_*]$ rule. Consider the last step in the proof of $\Gamma, (S \Rightarrow D \Leftarrow U) \to (C, L' \Rightarrow B' \Leftarrow T')$. It can not be an axiom because axioms are atomic. If it is a left rule, then we permute the left rule and the right rule. If it is a $[\backslash R_*]$ rule then we permute the two $[R_*]$ rules. If it is a $[/R_*]$ rule then we rearrange the steps in this sub-proof in the same way (the last step is a left rule or a $[\backslash R_*]$ rule). There must be a $[\backslash R_*]$ rule. You can rearrange the proof of $\Gamma, (S \Rightarrow D \Leftarrow U) \to (C, L' \Rightarrow B' \Leftarrow T')$ such that the last step is either a left rule or a $[\backslash R_*]$. The second case is treated in 1a.

Case 2a: The two R_* rules have the same direction. Let's repeat the cut rule:

$$\dfrac{\Gamma \to A \quad \Delta, A, \Delta' \to B}{\Delta, \Gamma, \Delta' \to B}\,[\text{Cut}]$$

There are two subcases: Δ' is empty or Δ' is not empty. When Δ' is not empty, you can just permute the right rule and the cut rule.

$$\dfrac{\dfrac{\cdots}{\Delta' \to A}\,[/R_*] \quad \dfrac{\Delta, A, \Delta'', (L' \Rightarrow B' \Leftarrow T') \to (L \Rightarrow B \Leftarrow T)}{\Delta, A, \Delta'', (L' \Rightarrow B' \Leftarrow T', C) \to (L \Rightarrow B \Leftarrow T, C)}\,[/R_*]}{\Delta, \Delta', \Delta'', (L' \Rightarrow B' \Leftarrow T', C) \to (L \Rightarrow B \Leftarrow T, C)}\,[\text{Cut}]$$

Transform this into:

$$\dfrac{\dfrac{\Delta' \to A \quad \Delta, A, \Delta''(L' \Rightarrow B' \Leftarrow T') \to (L \Rightarrow B \Leftarrow T)}{\Delta, \Delta', \Delta'', (L' \Rightarrow B' \Leftarrow T') \to (L \Rightarrow B \Leftarrow T)}\,[\text{Cut}]}{\Delta, \Delta', \Delta'', (L' \Rightarrow B' \Leftarrow T', C) \to (L \Rightarrow B \Leftarrow T, C)}\,[/R_*]$$

When Δ' in the cut rule is empty the proof looks like:

$$\dfrac{\dfrac{\Delta', (S \Rightarrow D \Leftarrow U) \to (L' \Rightarrow B' \Leftarrow T')}{\Delta', (S \Rightarrow D \Leftarrow U, A) \to (L' \Rightarrow B' \Leftarrow T', A)}\,[/R_*] \quad \dfrac{\Delta, (L' \Rightarrow B' \Leftarrow T') \to (L \Rightarrow B \Leftarrow T)}{\Delta, (L' \Rightarrow B' \Leftarrow T', A) \to (L \Rightarrow B \Leftarrow T, A)}\,[/R_*]}{\Delta, \Delta', (S \Rightarrow D \Leftarrow U, A) \to (L \Rightarrow B \Leftarrow T, A)}\,[\text{Cut}]$$

Transform this into:

$$\dfrac{\dfrac{\Delta', (S \Rightarrow D \Leftarrow U) \to (L' \Rightarrow B' \Leftarrow T') \quad \Delta, (L' \Rightarrow B' \Leftarrow T') \to (L \Rightarrow B \Leftarrow T)}{\Delta, \Delta', (S \Rightarrow D \Leftarrow U) \to (L \Rightarrow B \Leftarrow T)}\,[\text{Cut}]}{\Delta, \Delta', (S \Rightarrow D \Leftarrow U, A) \to (L \Rightarrow B \Leftarrow T, A)}\,[/R_*]$$

Case 2b: The two R_* rules do not have the same direction.

Like in 2a we can distinguish two cases: Δ' in the cut rule is empty or not. If it is not empty we can perform the same transformation as in case 2a. If Δ' is empty the proof has the following form:

$$\dfrac{\dfrac{\Delta' \to (L' \Rightarrow B' \Leftarrow T', A)}{\Delta' \to (C, L' \Rightarrow B' \Leftarrow T', A)}\,[\backslash R_*] \quad \dfrac{\Delta, (C, L' \Rightarrow B' \Leftarrow T') \to (L \Rightarrow B \Leftarrow T)}{\Delta, (C, L' \Rightarrow B' \Leftarrow T', A) \to (L \Rightarrow B \Leftarrow T, A)}\,[/R_*]}{\Delta, \Delta' \to (L \Rightarrow B \Leftarrow T, A)}\,[\text{Cut}]$$

Like in case 1b, we can change the proof of $\Delta' \to (C, L' \Rightarrow B' \Leftarrow T', A)$ such that the last step applied is a left rule or a $[/R_*]$ rule. The second case is treated in 2a. It is left to the reader to check that the degree of the Cut rule decreases in all proof transformations given here.

This finishes the proof of cut elimination for Aux.

3.4 The System ApplComp

Now we can introduce the calculus that is used in the recognition algorithm for second order Lambek based Categorial Grammar. The calculus is called ApplComp (short for Application and Composition). It has the following rules:

$$\mathsf{A} \to \mathsf{A}$$

$$\frac{\Delta, (\mathsf{U}, \mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{S}), \Delta' \to \mathsf{C}}{\Delta, (\mathsf{U}, \mathsf{V} \Rightarrow \mathsf{A} \Leftarrow \mathsf{W}), ((\mathsf{V} \Rightarrow \mathsf{A} \Leftarrow \mathsf{W}), \mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{S}), \Delta' \to \mathsf{C}}\ [\text{COMP}\backslash]$$

$$\frac{\Delta, (\mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{S}, \mathsf{U}), \Delta' \to \mathsf{C}}{\Delta, (\mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{S}, (\mathsf{V} \Rightarrow \mathsf{A} \Leftarrow \mathsf{W})), (\mathsf{V} \Rightarrow \mathsf{A} \Leftarrow \mathsf{W}, \mathsf{U}), \Delta' \to \mathsf{C}}\ [\text{COMP/}]$$

Again, A, B and C are types, Δ, Δ', S, T, U, V and W are (possibly empty) lists of types.

3.4.1. THEOREM. *ApplComp = Aux.*

We know that $Aux+[\text{Cut}] = Aux$. We are going to prove that $ApplComp \subseteq Aux+[\text{Cut}]$ and that $Aux \subseteq ApplComp$.

- $ApplComp \subseteq Aux + [\text{Cut}]$
  The following is a valid proof in Aux:
  $$\dfrac{\dfrac{(\mathsf{V} \Rightarrow \mathsf{A} \Leftarrow \mathsf{W}) \to (\mathsf{V} \Rightarrow \mathsf{A} \Leftarrow \mathsf{W}) \quad (\mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{S}) \to (\mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{S})}{(\mathsf{V} \Rightarrow \mathsf{A} \Leftarrow \mathsf{W}), ((\mathsf{V} \Rightarrow \mathsf{A} \Leftarrow \mathsf{W}), \mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{S}) \to (\mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{S})}\ [\backslash\text{L}]}{\dfrac{\dfrac{\dfrac{\cdots \Rightarrow \cdots}{\cdots \Rightarrow \cdots}\ [\backslash\text{R}_*]}{\cdots \Rightarrow \cdots}\ [\backslash\text{R}_*]}{(\mathsf{U}, \mathsf{V} \Rightarrow \mathsf{A} \Leftarrow \mathsf{W}), ((\mathsf{V} \Rightarrow \mathsf{A} \Leftarrow \mathsf{W}), \mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{S}) \to (\mathsf{U}, \mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{S})}\ [\backslash\text{R}_*]}$$
  In order to save space we call this conclusion Q. We can use Q in the following proof:
  $$\frac{Q \quad \Delta, (\mathsf{U}, \mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{S}), \Delta' \to \mathsf{C}}{\Delta, (\mathsf{U}, \mathsf{V} \Rightarrow \mathsf{A} \Leftarrow \mathsf{W}), ((\mathsf{V} \Rightarrow \mathsf{A} \Leftarrow \mathsf{W}), \mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{S}), \Delta' \to \mathsf{C}}\ [\text{Cut}]$$
  We have a proof of the [Comp\] rule in $Aux + [\text{Cut}]$.
- $Aux \subseteq ApplComp$
  - Suppose the last step is a left rule, e.g., [/L]. The rules of ApplComp have exactly one premise. Therefore, the ApplComp proof of the left premise of the [/L] rule of Aux looks like
    $$\begin{array}{c} A \to A \\ \vdots \\ \Gamma \to A \end{array}$$
    We can transform this into:
    $$\begin{array}{c} \Delta, (L \Rightarrow B \Leftarrow T, A), A, \Delta' \to C \\ \vdots \\ \Delta, (L \Rightarrow B \Leftarrow T, A), \Gamma, \Delta' \to C \end{array}$$
    and in
    $$\begin{array}{c} \dfrac{\Delta, (L \Rightarrow B \Leftarrow T), \Delta' \to C}{\Delta, (L \Rightarrow B \Leftarrow T, A), A, \Delta' \to C}\ [\text{COMP/}] \\ \vdots \\ \Delta, (L \Rightarrow B \Leftarrow T, A), \Gamma, \Delta' \to C \end{array}$$

– Suppose the last step is a right rule, e.g., [/R$_*$] The ApplComp proof of the premise of the [/R$_*$] rule looks like:

$$\begin{array}{c}(L \Rightarrow B \Leftarrow T) \rightarrow (L \Rightarrow B \Leftarrow T) \\ \vdots \\ \Gamma, (V \Rightarrow C \Leftarrow W) \rightarrow (L \Rightarrow B \Leftarrow T)\end{array}$$

This can be changed into:

$$\begin{array}{c}(L \Rightarrow B \Leftarrow T, A) \rightarrow (L \Rightarrow B \Leftarrow T, A) \\ \vdots \\ \Gamma, (V \Rightarrow C \Leftarrow W, A) \rightarrow (L \Rightarrow B \Leftarrow T, A)\end{array}$$

We conclude that the systems ApplComp, Aux and 2nd order Lambek calculus (L2) generate the same set of sequents.

3.5 The Algorithm

The algorithm we present here is based on the calculus ApplComp. It decides whether a sequent is derivable in ApplComp. When it is derivable, the algorithm finds all proofs in ApplComp of the sequent. The algorithm first constructs an initial graph called a *chart*. After the initialization, ϵ-arcs are added (ϵ is the empty word). These ϵ-arcs short-circuit paths in the graph. The set of vertices does not change after the initialization phase.

Beside the primitive types Pr and the types Tp we introduce the arrow-types ATp.

3.5.1. DEFINITION. If A is in Pr, then $\overset{\uparrow}{A}$ is in ATp. If A is in Tp, then $\overset{\leftarrow}{A}$ and $\vec{A}$ are in ATp.

With Tp^* and ATp^* we mean sequences of types and arrow types respectively. We define a mapping f :

3.5.2. DEFINITION. f: $Tp \rightarrow ATp^*$, the unfolding function, is defined as follows: $f((\mathsf{T}_1, \ldots, \mathsf{T}_m \Rightarrow \mathsf{B} \Leftarrow \mathsf{L}_1, \ldots, \mathsf{L}_n)) = \overset{\leftarrow}{T_1}, \ \ldots \ \overset{\leftarrow}{T_m}, \ \overset{\uparrow}{B}, \ \overset{\rightarrow}{L_1}, \ \ldots \ \overset{\rightarrow}{L_n} \quad (m, n \geq 0)$. We extend the definition to $f : Tp^* \rightarrow ATp^*$ by applying f pointwise and concatenating the result.

We construct the initial chart as follows. Given the input sentence we look up the types assigned to the words in the lexicon. For every type X we put a path $f(X)$ in the chart. The chart contains two types of vertices. The first type, drawn as a little square, separates the words in the sentence. The second type, drawn as a dot, separates the elements of $f(X)$. The edges are the subtypes of the types of the words in the sentence.

Lexical ambiguities cause multiple paths between square vertices. This can be seen in Figure 3.5.

Observe that complex arguments appear in the chart as arcs. They are not decomposed into smaller parts. In this respect the unfolding differs from the proofnet unfolding found in e.g. (Roorda 1991). We construct a rewrite grammar with the following two types of rules:

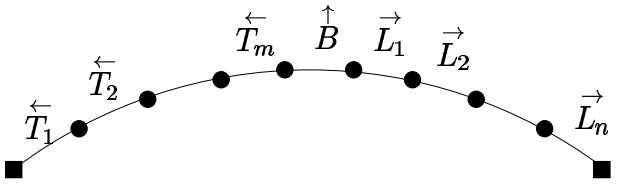


Figure 3.4: Unfolding.

$$\epsilon \rightarrow (\mathsf{T}_1, \ldots, \mathsf{T}_m \Rightarrow \overrightarrow{\mathsf{B}} \Leftarrow \mathsf{L}_1, \ldots, \mathsf{L}_n), \overleftarrow{T}_1, \ \ldots \ \overleftarrow{T}_m, \ \overrightarrow{B}, \ \overrightarrow{L}_1, \ \ldots \ \overrightarrow{L}_n \quad (m, n \geq 0)$$

$$\epsilon \rightarrow \overleftarrow{T}_1, \ \ldots \ \overleftarrow{T}_m, \ \overrightarrow{B}, \ \overrightarrow{L}_1, \ \ldots \ \overrightarrow{L}_n, \ (\mathsf{T}_1, \ldots, \mathsf{T}_m \Rightarrow \overleftarrow{\mathsf{B}} \Leftarrow \mathsf{L}_1, \ldots, \mathsf{L}_n)$$

where the type $(\mathsf{T}_1, \ldots, \mathsf{T}_m \Rightarrow \mathsf{B} \Leftarrow \mathsf{L}_1, \ldots, \mathsf{L}_n)$ ranges over all argument types in the lexicon. All T_i and L_i are primitive. ϵ is a special symbol denoting the empty word. The size of the grammar is linear in the size of the lexicon. When we add ϵ-arcs in the chart, (sub-)types cancel each other out, and we shortcircuit paths.

The algorithm does the following: find bottom-up all ϵ-arcs. We follow the standard bottom up algorithms for context free grammar recognition (Winograd 1983). Intermediate results are stored in so-called *pending edges* of the form

$$\epsilon \rightarrow \boxed{i} \ \overleftarrow{T}_1, \ \ldots \ \overleftarrow{T}_m, \ \overrightarrow{B} \ \boxed{j}, \ \overrightarrow{L}_1, \ \ldots \ \overrightarrow{L}_n, \ (\mathsf{T}_1, \ldots, \mathsf{T}_m \Rightarrow \overleftarrow{\mathsf{B}} \Leftarrow \mathsf{L}_1, \ldots, \mathsf{L}_n)$$

The superscripts i and j indicate labels of vertices. The position of j is the position of the "dot" well-known from chart parsing (do not confuse with dot vertices in the chart!). In this example there is a path $\overleftarrow{T}_1, \ \ldots \ \overleftarrow{T}_m, \ \overrightarrow{B}$ between the vertices i and j. There are two possibilities to move the dot.

1. When we find an ϵ from j to k we add the pending edge:

$$\epsilon \rightarrow \boxed{i} \ \overleftarrow{T}_1, \ \ldots \ \overleftarrow{T}_m, \ \overrightarrow{B} \ \boxed{k}, \ \overrightarrow{L}_1, \ \ldots \ \overrightarrow{L}_n, \ (\mathsf{T}_1, \ldots, \mathsf{T}_m \Rightarrow \overleftarrow{\mathsf{B}} \Leftarrow \mathsf{L}_1, \ldots, \mathsf{L}_n)$$

2. When we find an $\overrightarrow{L}_1$ from j to k we add the pending edge:

$$\epsilon \rightarrow \boxed{i} \ \overleftarrow{T}_1, \ \ldots \ \overleftarrow{T}_m, \ \overrightarrow{B}, \ \overrightarrow{L}_1 \ \boxed{k}, \ \ldots \ \overrightarrow{L}_n, \ (\mathsf{T}_1, \ldots, \mathsf{T}_m \Rightarrow \overleftarrow{\mathsf{B}} \Leftarrow \mathsf{L}_1, \ldots, \mathsf{L}_n)$$

The algorithm proceeds from left to right. All dot vertices between two square vertices are considered before the algorithm proceeds to the next word. The algorithm terminates when it has reached the final vertex, behind the last word. Only when there is a path $\epsilon, \ldots, \epsilon, S, \epsilon, \ldots, \epsilon$ spanning the whole input sentence, the sentence is correct.

We will now give an example. Suppose our lexicon is the example lexicon of page 30 and that the input sentence is "everyone loves somebody".

everyone	pn
	$(s \Leftarrow (np \Rightarrow s))$
loves	$(np \Rightarrow s \Leftarrow np)$
somebody	pn
	$((s \Leftarrow np) \Rightarrow s)$

Two important grammar rules are:

$\epsilon \rightarrow (np \overrightarrow{\Rightarrow} s),\ \overleftarrow{np},\ \overset{\uparrow}{s}$

$\epsilon \rightarrow \overset{\uparrow}{s},\ \overrightarrow{np},\ (s \overleftarrow{\Leftarrow} np)$

The initial chart is in Figure 3.5.

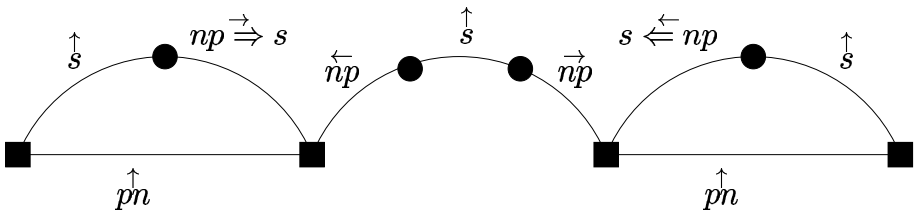


Figure 3.5: Initial chart.

Figure 3.6 shows which ϵ-arcs are added when the grammar rules are applied.

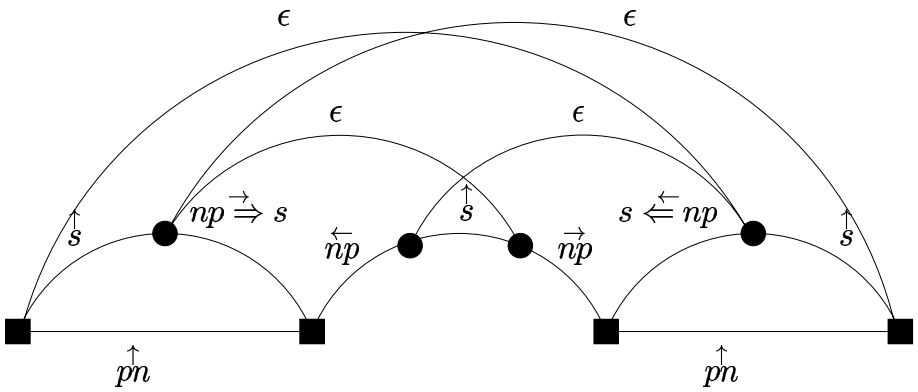


Figure 3.6: Final chart.

We see that the algorithm finds two paths labeled s from the begin vertex to the end vertex. One path is $\overset{\uparrow}{s}\ \epsilon$, the other $\epsilon\ \overset{\uparrow}{s}$. These two paths correspond to the two readings we should obtain.

3.6 Proof of Correctness of the Algorithm

Define *LexTypes* as the set of all subtypes in lexicon:
$Lextypes = \{Lt_i | \exists w, t((w,t) \in Lex \wedge f(t) = \ldots, Lt_i, \ldots)\}$ We will use the abbreviation $\overleftarrow{p}$ for $\{\overleftarrow{A} \mid \overleftarrow{A} \in Lextypes\}$, $\overset{\uparrow}{p}$ for $\{\overset{\uparrow}{A} \mid \overset{\uparrow}{A} \in Lextypes\}$ and $\overrightarrow{p}$ for $\{\overrightarrow{A} \mid \overrightarrow{A} \in Lextypes\}$. With a path through the chart we mean a path from left to right with the ϵ-arcs left out.

3.6.1. LEMMA. *Any path through the chart from begin vertex to end vertex is a member of the regular language:*

$$((\overleftarrow{p})^* \overset{\uparrow}{p} (\overrightarrow{p})^*)^*$$

Proof: The Lemma obviously holds for the paths in the initial chart. Application of a rule adds an ϵ-arc in the chart. The grammar rules look like:

$$\epsilon \rightarrow \overrightarrow{p} (\overleftarrow{p})^* \overset{\uparrow}{p} (\overrightarrow{p})^*$$

$$\epsilon \rightarrow (\overleftarrow{p})^* \overset{\uparrow}{p} (\overrightarrow{p})^* \overleftarrow{p}$$

If the lemma holds before application of a rule it will still hold afterwards. □

3.6.2. DEFINITION. Define derivability $\Rightarrow$ between two lists of types X and Y as follows (s is the distinguished type of the categorial grammar):

X $\Rightarrow$ Y iff

$$\frac{X \rightarrow s}{Y \rightarrow s} \text{ [COMP/]}$$

or

$$\frac{X \rightarrow s}{Y \rightarrow s} \text{ [COMP\]}$$

The transitive reflexive closure of $\Rightarrow$ is $\Rightarrow^*$

Any path from the begin vertex to the end vertex through the chart can be translated in a list of types X by simply applying f^{-1}. (see also lemma 3.6.1).

The invariant that is maintained in the algorithm is the following. For all the X that are the result of applying f^{-1} to some path from begin vertex to end vertex in the chart, if the input sentence is $a_1 \ldots a_n$ then there are some types $A_1, \ldots, A_n, \langle a_i, A_i \rangle \in Lex(1 \leq i \leq n)$ such that $X \Rightarrow^* A_1, \ldots, A_n$.

If the start symbol s is among these X's then the input sentence is grammatical.

Addition of an ϵ arc in the chart corresponds exactly with application of the composition rule of ApplComp. The ϵ-rule is:

$$\epsilon \rightarrow \overleftarrow{V_1}, \ \ldots \ \overleftarrow{V_m}, \ \overset{\uparrow}{A}, \ \overrightarrow{W_1}, \ \ldots \ \overrightarrow{W_n}, \ (\mathsf{V_1}, \ldots, \mathsf{V_m} \Rightarrow \overleftarrow{\mathsf{A}} \Leftarrow \mathsf{W_1}, \ldots, \mathsf{W_n})$$

The composition rule is:

$$\frac{\Delta, (\mathsf{U}, \mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{S}), \Delta' \rightarrow \mathsf{C}}{\Delta, (\mathsf{U}, \mathsf{V} \Rightarrow \mathsf{A} \Leftarrow \mathsf{W}), ((\mathsf{V} \Rightarrow \mathsf{A} \Leftarrow \mathsf{W}), \mathsf{T} \Rightarrow \mathsf{B} \Leftarrow \mathsf{S}), \Delta' \rightarrow \mathsf{C}} \text{ [COMP\]}$$

For any path

$$\Delta \blacksquare \overleftarrow{U} \bullet \overleftarrow{V} \bullet \overset{\uparrow}{A} \bullet \overrightarrow{W} \blacksquare (V \Rightarrow \overleftarrow{A} \Leftarrow W) \bullet \overleftarrow{T} \bullet \overset{\uparrow}{B} \bullet \overrightarrow{S} \blacksquare \Delta'$$

a new path

$$\Delta \blacksquare \overleftarrow{U} \bullet \epsilon \bullet \overleftarrow{T} \bullet \overset{\uparrow}{B} \bullet \overrightarrow{S} \blacksquare \Delta'$$

is added. The invariant is maintained and the algorithm finds all X's with the desired property.

The algorithm is polynomial time because there is a polynomial number of vertices and therefore a polynomial number of arcs, labeled ϵ, and a polynomial number of pending edges is added.

Section 7.2 gives an alternative algorithm that is polynomial time too. It is based on a method to estimate time complexity which is described in chapter 5.

3.7 Discussion

We first show that the method presented here does not work in higher order fragments (> 2). The type $(((c \Rightarrow c) \Rightarrow b) \Rightarrow b)$ has order 3. Consider the sequent:

$$(c \Rightarrow b), ((c \Rightarrow b) \Rightarrow b), (((c \Rightarrow c) \Rightarrow b) \Rightarrow b) \rightarrow b$$

This sequent is derivable in the Lambek calculus. But it is not derivable in ApplComp. There are only arrows to the right so the only applicable rule is the following simplified ApplComp rule:

$$\frac{\Delta, (\mathsf{U}, \mathsf{T} \Rightarrow \mathsf{B}), \Delta' \rightarrow \mathsf{C}}{\Delta, (\mathsf{U}, \mathsf{V} \Rightarrow \mathsf{A}), ((\mathsf{V} \Rightarrow \mathsf{A}), \mathsf{T} \Rightarrow \mathsf{B}), \Delta' \rightarrow \mathsf{C}} \text{ [COMP\ simple]}$$

It is clear that the sequent is not derivable.

The second order fragment has been studied earlier, not only by Buszkowski (1990) but also by Hepple (1991) and Barry (1992). They use natural deduction style proofs instead of Gentzen style proofs. In natural deduction, the second order restriction is equivalent to the restriction that hypotheses in the proof are atomic.

We can define the notion *virtual* second order as follows. A lexicon is virtual second order if it can be made into a real second order lexicon under some renaming

of subtypes. All the results presented in this paper are valid for virtual second order Lambek based categorial grammar.

In natural deduction, virtual second order means that assumptions in proofs are arguments and not functors. The system D of Barry (1992) is defined like this: hypotheses must be arguments. Therefore, virtual second order is precisely D^2.

Furthermore the second order Lambek calculus seems, at first sight, to be equivalent with the Lambek derivable fragment of Steedman's Combinatory Categorial Grammar. In CCG, only application and composition are allowed. However, CCG has no associativity rules. Composition of $w\backslash y, y\backslash(x/z)$ into $w\backslash(x/z)$ is possible but composition of $w\backslash y, (y\backslash x)/z$ into $w\backslash(x/z)$ is not. If we allow for lifting as an operation in the lexicon we can get an equivalent system.

[2]There is a small problem with associativity: $x/((b\backslash a)/c), b\backslash(a/c) \rightarrow x$ is not derivable in D. It is derivable in virtual second order Lambek calculus when we use the "flat" notation. Hepple (1991) argues that associativity should be added to D.

Chapter 4

Acyclic Context-sensitive Grammars

In this chapter[1] we propose a new type of context-sensitive grammars, the *acyclic* context-sensitive grammars (ACSG's). Acyclic context-sensitive grammars are context-sensitive grammars with rules that have a "real rewrite part", which must be context-free and acyclic, and a "context part". The context is present on both sides of a rule. The motivation for the introduction of ACSG's is that we want a formalism which

- allows parse trees with crossing branches
- is computationally tractable
- has a simple definition.

We enrich context-free rewrite rules with context and this leads to a simple definition of a formalism that generates parse trees with crossing branches. In order to gain efficiency, we add a restriction: the context-free rewrite part of the grammar must be acyclic. Without this restriction, the complexity would be the same as for unrestricted CSG's (PSPACE-complete). With the acyclicity restriction we get the same results as for *growing* CSG's (Dahlhaus and Warmuth 1986, Buntrock 1993), i.e., NP-completeness for uniform recognition and polynomial time for fixed grammars.

Possible applications are in the field of computational linguistics. In natural language one often finds sentences with so-called *discontinuous constituents* (constituents separated by other material). ACSG's can be used to describe such constructions. Most similar attempts (Pereira 1981, Johnson 1985, Bunt 1988, Abramson and Dahl 1989) allow an arbitrary distance between two parts of a discontinuous constituent. This is not allowed in ACSG's. For unbounded dependencies like *wh-movement* we either have to extend the formalism (allow arbitrary context) or introduce the *slash-feature*.

The acyclicity of the grammar does not seem to form a problem for the generative capacity necessary to describe natural language. Acyclicity is closely related to the *off-line parsability constraint* (Johnson 1988). Constituent structures satisfy the off-line parsability constraint iff

- they do not include a non-branching dominance chain in which the same category appears twice, and
- the empty string ϵ does not appear as the righthand side of any rule.

[1] The NP-completeness proof in this chapter was published in Aarts (1992).

The off-line parsability constraint has been motivated both computationally and from the linguistic perspective. Kaplan and Bresnan (1982) say that "vacuously repetitive structures are without intuitive or empirical motivation" (Johnson 1988). ACSG's satisfy the off-line parsability constraint: they have no cycles and no ϵ-rules.

The goal of designing a formalism that is computationally tractable is only achieved partially. We show that the uniform recognition problem is NP-complete. For any fixed grammar, however, the recognition problem is polynomial (in the length of the sentence).

The definition of ACSG is simple because it is a standard rewrite grammar. Derivability is defined by successive string replacement. This definition is, e.g., simpler than the definition of Discontinuous Phrase Structure Grammar (Bunt 1988). The main difference between DPSG and ACSG is that in ACSG constituents are "moved" when a context-sensitive rule is applied. In DPSG trees with crossing branches are described "staticly": the shape of a tree is described by node admissibility constraints.

The structure of this chapter is as follows. First we define acyclic CSG's, growing CSG's and quasi-growing CSG's formally. Then we present some results on the generative power of these classes of grammars and on their time complexity. We end with a discussion on the uniform recognition problem vs. the recognition problem for fixed grammars.

4.1 Definitions

4.1.1 Context-sensitive Grammars

4.1.1. DEFINITION. *A* grammar *is a quadruple $\langle V, \Sigma, S, P\rangle$, where V is a finite set of symbols and $\Sigma \subset V$ is the set of terminal symbols. Σ is also called the* alphabet. *The symbols in $V \setminus \Sigma$ are called the nonterminal symbols. $S \in V \setminus \Sigma$ is a start symbol and P is a set of production rules of the form $\alpha \to \beta$, with $\alpha, \beta \in V^*$, where α contains at least one nonterminal symbol.*

4.1.2. DEFINITION. *A grammar is* context-free *if each rule is of the form $Z \to \gamma$ where $Z \in V \setminus \Sigma$; $\gamma \in V^*$. A* cycle *in a context-free grammar is a set of symbols $a_1, \ldots, a_n$ with $a_i \to a_{i+1} \in P$ for all $1 \leq i < n$ and $a_1 = a_n$.*

4.1.3. DEFINITION. *A grammar is* context-sensitive *if all rules are of the form $\alpha \to \beta$, with $|\alpha| \leq |\beta|$.*

4.1.4. DEFINITION. Derivability *($\Rightarrow$) between strings is defined as follows: $u\alpha v \Rightarrow u\beta v$ $(u, v, \alpha, \beta \in V^*)$ iff $(\alpha, \beta) \in P$. The transitive closure of $\Rightarrow$ is denoted by $\stackrel{+}{\Rightarrow}$. The reflexive transitive closure of $\Rightarrow$ is denoted by $\stackrel{*}{\Rightarrow}$.*

4.1.5. DEFINITION. *The* language *generated by G is defined as $L(G) = \{w \in \Sigma^* \mid S \stackrel{*}{\Rightarrow} w\}$.*

4.1.6. DEFINITION. *A derivation of a string δ is a sequence of strings $x_1, x_2, \ldots, x_n$ with $x_1 = S$, for all i $(1 \leq i < n)$ $x_i \Rightarrow x_{i+1}$ and $x_n = \delta$.*

4.1.2 Labeled Context-sensitive Grammars

Context-sensitive grammars have just been described as grammars with rules of the form $\alpha \to \beta$ for which $|\alpha| \leq |\beta|$. There is an alternative definition where *context* is used. In this definition, rules are of the form $\alpha Z \beta \to \alpha\gamma\beta$ ($Z \in V, \alpha, \beta, \gamma \in V^*$). In this format, α and β are *context*, and $Z \to \gamma$ is called the *context-free backbone*. It is known that the two formats are weakly equivalent (Salomaa 1973, pp. 15,82). We introduce here a third form, which is a kind of a mix between the other two. Instead of having context on the outside of the rule, we can also have context inside. Suppose we have the rule "A rewrites to B C in the context D". In a standard context-sensitive grammar, this can only be expressed as D A $\to$ D B C or as A D $\to$ B C D. We are going to allow that the D is in between the B and the C. A possible rule is D A $\to$ B D C. In the rules of the form $\alpha Z \beta \to \alpha\gamma\beta$ it is not always clear which symbols are the context and which symbols form the context-free backbone. E.g., in the grammar rule A A $\to$ A B A it is not clear what the context is. We are going to indicate the context with brackets. The rule A A $\to$ A B A can be written as [A] A $\to$ [A] B A or as A [A] $\to$ A B [A] (the context is between brackets). In the new form, where the context can be "scattered", brackets are not enough. Therefore we introduce labels for the context symbols. The labels left and right of the arrow must be the same of course. The rule [A] A $\to$ [A] B A is written now as A_1 A $\to$ A_1 B A and A [A] $\to$ A B [A] is written as A A_1 $\to$ A B A_1. The rule D A $\to$ B D C can be written as D_1 A $\to$ B D_1 C.

This can be formalized as follows.

4.1.7. DEFINITION. *An* acyclic context-sensitive grammar *G is a quadruple $\langle V, \Sigma, S, P\rangle$, where V and Σ are, again, sets of symbols and terminal symbols. V_l and Σ_l (labeled symbols and terminals) denote $\{\langle a, k\rangle \mid a \in V, 1 \leq k \leq K\}$ and $\{\langle a, k\rangle \mid a \in \Sigma, 1 \leq k \leq K\}$ respectively, where K depends on the particular grammar under consideration. $S \in V \setminus \Sigma$ is a start symbol and P is a set of production rules of the form $\alpha \to \beta$, with $\alpha \in V_l^* V V_l^*, \beta \in (V \cup V_l)^*$. The left hand side contains exactly one unlabeled symbol, the right hand side at least one. For all production rules it holds that $|\alpha| \leq |\beta|$. The labeled symbols in a rule are called the context.*

There are three conditions:

- *If we leave out all members of V_l (the context) from the production rules we obtain a context-free grammar. This is the context-free backbone.*
- *If we leave out all members of V (the backbone) from the production rules we obtain a grammar that has permutations only. This is the context part. All context symbols in the rules should have different labels.*
- *If we remove all labels in the rules, i.e., we replace all symbols $\langle a, b\rangle$ by a, we get an ordinary context-sensitive grammar G'. We define that $L(G) = L(G')$.*

4.1.3 Acyclic Context-sensitive Grammars

Acyclic context-sensitive grammars, or ACSG's, are context-free grammars with an "acyclic contextfree backbone". This is formalized as follows.

4.1.8. DEFINITION. *An* acyclic context-sensitive grammar *is a labeled context-sensitive grammar that fullfills the following condition:*

- *If we leave out all members of V_l from the production rules we must obtain a finitely ambiguous context-free grammar, i.e. a grammar for which $|\alpha| = 1$ and $|\beta| \geq 1$ and that contains no cycles.*

An example of a rule of an acyclic CSG is (pairs of $\langle N, L \rangle$ are written as N_L):

$$A_1\ B_2\ M\ C_3\ D_4 \rightarrow C_3\ K\ B_2\ A_1\ L\ M\ D_4$$

The context-free backbone is M $\rightarrow$ K L M. The context part is $A_1\ B_2\ C_3\ D_4 \rightarrow C_3\ B_2\ A_1\ D_4$. The rule without labels is A B M C D $\rightarrow$ C K B A L M D. Another example is given after the definition of growing context-sensitive grammars.

4.1.4 Growing Context-sensitive Grammars

The definition of growing CSG's (GCSG's) is pretty simple: the lefthand side of a rule must be shorter than the righthand side. For the precise definition we follow Buntrock (1993):

4.1.9. DEFINITION. *A context-sensitive grammar $G = \langle V, \Sigma, S, P \rangle$ is growing if*

1. $\forall(\alpha \rightarrow \beta) \in P : \alpha \neq S \Rightarrow |\alpha| < |\beta|$, *and*
2. *S does not appear on the right hand side of any rule.*

We also define grammars which are growing with respect to a weight function. These were introduced in (Buntrock 1993).

4.1.10. DEFINITION. *We call a function $f : \Sigma^* \rightarrow \mathbb{N}$ a* weight function *if $\forall a \in \Sigma : f(a) > 0$ and $\forall w, v \in \Sigma^* : f(w) + f(v) = f(wv)$.*

Now we can define *quasi-growing* grammars:

4.1.11. DEFINITION. *Let $G = \langle V, \Sigma, S, P \rangle$ be a grammar and $f : V^* \rightarrow \mathbb{N}$ a weight function. We call G quasi-growing if $f(\alpha) < f(\beta)$ for all productions $(\alpha \rightarrow \beta) \in P$.*

Quasi-growing context-sensitive grammars (QGCSG's) are grammars that are both quasi-growing and context-sensitive.

4.2 An Example

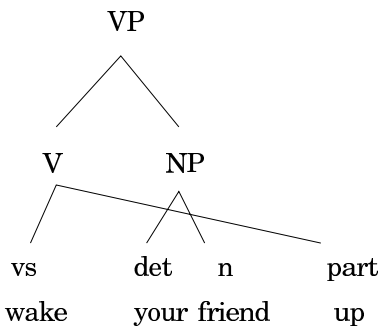
This section contains an example based on natural language of how ACSG's work. It is taken from Bunt (1988). Assume we have the following grammar. It has one rule that is not context-free.

Suppose we have the sentence: "Wake your friend up". This is a VP:
VP $\Rightarrow$ V NP $\Rightarrow$ vs NP part $\Rightarrow$ vs det n part $\Rightarrow^*$ wake your friend up

VP	→	V	NP		vs	→	wake
V NP_1	→	vs	NP_1	part	det	→	your
NP	→	det	n		n	→	friend
					part	→	up

Table 4.1: An example grammar

The corresponding parse tree is:



We see that we have a context-free backbone which allows us to draw parse trees (or structure trees) and that the scattered context causes branches in the trees to cross.

4.3 Properties of Acyclic Context-sensitive Grammars

4.3.1 Generative Power of Acyclic CSG's

In this section we discuss the relation between ACSG's and GCSG's and we sketch their position in the Chomsky hierarchy.

4.3.1. THEOREM. *ϵ-free CFL $\subset$ ACSL*

Proof: if the cfl is generated by an acyclic cfg without empty productions we do not have to do anything. This cfg *is* an acyclic csg. If the cfg contains cycles we can remove them. A cycle can be removed by the introduction of a new symbol. This symbol rewrites to any member of the cycle. Any cfg with empty productions can be changed into a cfg without empty productions that generates the same language. There's one exception here: languages containing the empty string can not be generated. Therefore acyclic context-sensitive grammars generate all cfl's that do not contain the empty word. □

4.3.2. THEOREM. *ACSL $\neq$ CFL*

Proof: ACSG's are able to generate languages that are not context-free. One example is the language $\{a^n b^{2^n} c^n \mid n \geq 1\}$. This language is generated by the grammar:

A derivation of " a a b b b b c c " is:

S	$\rightarrow$	A B B C	A	$\rightarrow$	a
B X_1	$\rightarrow$	X_1 B B	B	$\rightarrow$	b
B C_1	$\rightarrow$	X B B C C_1	C	$\rightarrow$	c
A_1 X B_2	$\rightarrow$	A_1 A B_2			

Table 4.2: Grammar for $\{a^n b^{2^n} c^n \mid n \geq 1\}$

S $\Rightarrow$ A B B C $\Rightarrow$ A B X B B C C $\Rightarrow$ A X B B B B C C $\Rightarrow$ A A B B B B C C
$\stackrel{*}{\Rightarrow}$ a a b b b b c c.

We see that together with an A an X is generated. This X is sent through the sequence of B's in the middle. When the X meets the C's on the right-hand side it is changed into a C. While the X travels through the B's, the number of B's is doubled. This is different from an ordinary CSG. In an ordinary CSG it is possible to have a travelling X that does not double the material it passes (with a rule like
X B $\rightarrow$ B X). □

4.3.3. LEMMA. *ACSL* $\subset$ *QGCSL*

Suppose we have an acyclic context-sensitive grammar $G = (\langle V, \Sigma, S, P \rangle)$. We construct a QGCSG $G' = (\langle V, \Sigma, S, P' \rangle)$ with weight function g that generates the same language as the ACSG as follows. P' is obtained by removing all labels from P. G and G' generate the same language by definition (the weight function is irrelevant for the generative capacity).

It remains to show that we can construct a function g such that $\forall \alpha \rightarrow \beta \in P' : g(\alpha) < g(\beta)$. First we construct a graph as follows. For every unlabeled symbol in the ACSG there is a vertex in the graph. There is an arc from vertex T to vertex U iff the unary rule $T \rightarrow U$ is in the context-free backbone of the grammar. With $|V|$ we denote the cardinality of a set V. The maximal number of vertices in the graph is $|V|$.

We introduce a counter i that is initialized to $|V| + 1$. Assign the weight i to all vertices that are not connected to any other vertex and remove them. Now search the graph for a vertex without incoming edges. The weight of this symbol is i. Remove the vertex. Increment i by 1 and search for the next vertex without incoming edges. Repeat this until the graph is empty. The algorithm is guaranteed to stop with an empty graph because the graph is acyclic. Observe that $\forall x \in V : |V| < g(x) \leq 2|V|$.

We have to prove now that $\forall \alpha \rightarrow \beta \in P' : g(\alpha) < g(\beta)$. We consider two cases:

- $|\alpha| = |\beta|$. In this case the context-free backbone of the ACSG rule is unary. The weight of the context symbols is irrelevant because they occur both in α *and* in β. The context-free backbone is of the form $A \rightarrow B$. We know that $g(A) < g(B)$ because A was removed earlier from the graph than B and therefore has a lower weight.
- $|\alpha| < |\beta|$. Again, the weight of the context symbols is irrelevant because they are equal both left and right of the arrow. The context-free backbone is of the form $Z \rightarrow \gamma$, with $|\gamma| > 1$. We know that $g(Z) \leq 2|V|$ and that $g(\gamma) > 2|V|$, hence $g(Z) < g(\gamma)$. □

4.3.4. LEMMA. *QGCSL ⊂ GCSL*

The proof is in Buntrock and Loryś (1992) and is repeated here. For any QGCSG G we construct a GCSG G' and a homomorphism h such that $L(G') = h(L(G))$. Then we introduce a GCSG G'' with $L(G'') = L(G)$. We cite Buntrock and Loryś (1992):

> Let L be generated by a quasi-growing grammar $G = \langle V, \Sigma, S, P\rangle$ with weight function f. Without loss of generality we can assume that S does not appear on the right hand side of any production and that $f(S) = 1$. Let $c \notin V$ and let h be a homomorphism such that $h(a) = ac^{f(a)-1}$ for each $a \in V$. Then $G' = \langle V \cup \{c\}, \Sigma, S, P'\rangle$, where $P' = \{h(\alpha) \to h(\beta) : (\alpha \to \beta) \in P\}$ is a growing context-sensitive grammar and $L(G') = h(L(G))$.

GCSL's are closed under inverse homomorphism (Buntrock and Loryś 1992). Therefore there exists a GCSG G'' that recognizes $L(G)$. □

4.3.5. THEOREM. *ACSL ⊂ GCSL*

Follows immediately from Lemma's 4.3.3 and 4.3.4.

It is not known whether GCSL ⊂ ACSL. Buntrock and Loryś (1992) show that GCSL ≠ CSL. We get the following dependencies:

CFL ⊂ ACSL ⊆ GCSL ⊂ CSL

4.3.2 Complexity of Acyclic CSG's

We formally introduce the following problems:

UNIFORM RECOGNITION FOR ACYCLIC CONTEXT-SENSITIVE GRAMMAR
INSTANCE: An acyclic context-sensitive grammar $G = (V, \Sigma, S, P)$ and a string $w \in \Sigma^*$.
QUESTION: Is w in the language generated by G ?

UNIFORM RECOGNITION FOR GROWING CONTEXT-SENSITIVE GRAMMAR
INSTANCE: A growing context-sensitive grammar $G = (V, \Sigma, S, P)$ and a string $w \in \Sigma^*$.
QUESTION: Is w in the language generated by G ?

RECOGNITION FOR ACYCLIC CONTEXT-SENSITIVE GRAMMAR G
INSTANCE: A string $w \in \Sigma^*$.
QUESTION: Is w in the language generated by G ?

RECOGNITION FOR GROWING CONTEXT-SENSITIVE GRAMMAR G
INSTANCE: A string $w \in \Sigma^*$.
QUESTION: Is w in the language generated by G ?

We can prove two theorems:

4.3.6. THEOREM. *There is a polynomial time reduction from UNIFORM RECOGNITION FOR ACYCLIC CONTEXT-SENSITIVE GRAMMAR to UNIFORM RECOGNITION FOR GROWING CONTEXT-SENSITIVE GRAMMAR*

4.3.7. THEOREM. *For every G there is a G′, such that RECOGNITION FOR ACYCLIC CONTEXT-SENSITIVE GRAMMAR G is polynomial time reducible to RECOGNITION FOR GROWING CONTEXT-SENSITIVE GRAMMAR G′*

Proof of both theorems. Consider the proofs of Lemma's 4.3.3 and 4.3.4. These proofs show how we can find for every ACSG G a GCSG G' and a homomorphism h such that $L(G') = h(L(G))$. We know that $w \in L(G)$ iff $h(w) \in L(G')$. The reduction is polynomial time. This can be seen as follows. Computing a weight function for an ACSG costs kwadratic time. The weights are linear in $|G|$. Computation of h and G' from the weight function is linear. The total reduction is quadratic. Observe that the reduction from QGCSG to GCSG is not polynomial in general. We can represent the values of the weight function with a number of bits that is logarithmic in the value. The number of special symbols c added equals the value of the weight function, therefore the GCSG is exponential in the size of the QGCSG plus the weight function. But in the reduction here, the value of the weight function is linear in the size of the grammar, which makes the reduction polynomial.□

4.3.8. THEOREM. *The problem RECOGNITION FOR ACYCLIC CONTEXT-SENSITIVE GRAMMAR G is in P for all G.*

Dahlhaus and Warmuth (1986) proved that RECOGNITION FOR GROWING CONTEXT-SENSITIVE GRAMMAR G' is in P for all G'. The theorem follows from their proof and from Theorem 4.3.7. Aarts (1991) conjectured that the fixed grammar recognition problem was in P. An algorithm was given there without an estimate of the time complexity.

4.3.9. THEOREM. *The problem UNIFORM RECOGNITION FOR ACYCLIC CONTEXT-SENSITIVE GRAMMAR is NP-complete.*

This is proved in the next section.

4.3.10. THEOREM. *The problem UNIFORM RECOGNITION FOR GROWING CONTEXT-SENSITIVE GRAMMAR is NP-complete.*

From Theorems 4.3.6 and 4.3.9 it follows that the uniform recognition problem for GCSG is NP-hard. The problem is also in NP because we can guess derivations in GCSG and derivations in GCSG are very short (their length is smaller than the size of the input). □

This result is not new, however. The problem was put forward in an article from Dahlhaus and Warmuth (1986) and has been solved three times (Buntrock and Lorýs 1992).

4.4 Uniform Recognition for ACSG is NP-complete

In this section we prove that the uniform recognition problem is NP-complete. First we prove that the class of acyclic csg's is a subset of the class of linear time csg's, i.e., we prove that derivations are short. This implies that the uniform recognition problem for ACSG is in NP. Furthermore we give a reduction from 3-SAT to uniform ACSG recognition which completes the proof of NP-completeness of the problem.

For an introduction to the problem "3-SAT" the reader is referred to (Garey and Johnson 1979). Before we prove that UNIFORM RECOGNITION FOR ACYCLIC CONTEXT-SENSITIVE GRAMMAR is NP-complete, we first prove some theorems and lemmas.

Suppose $G' = (V', \Sigma', S', P')$ is an acyclic *context-free* grammar. The function $ld(G'', n)$ is the **l**ongest **d**erivation from any input word with length n ($n \geq 1$) using grammar G''.

4.4.1. LEMMA. $ld(G', n) \leq (2|P'| + 1)n$

Proof: With induction to n.

Basic step: $n = 1$. In the worst case we can apply all rules once. The length of this derivation is $|P'|$. So $ld(G', 1) = |P'|$.

Induction step.

Suppose $ld(G', m) \leq (2|P'| + 1)m$ for all $m < n$ $(n > 1)$. Suppose we have a derivation of a string of length n. This derivation has the form
$s \Rightarrow^* a_1 \dots a_i z a_{i+1} \dots a_{n-j} \Rightarrow a_1 \dots a_i z_1 \dots z_j a_{i+1} \dots a_{n-j} \Rightarrow^* b_1 \dots b_n$ ($j \geq 2$), where $z \rightarrow z_1 \dots z_j$ is the last growing rule. Afterwards only unary rules are applied. We can permute applications of some unary rules with application of the rule $z \rightarrow z_1 \dots z_j$ now, such that we get a derivation of the form
$s \Rightarrow^* b_1 \dots b_i z b_{i+j+1} \dots b_n \Rightarrow b_1 \dots b_i z_1 \dots z_j b_{i+j+1} \dots b_n \Rightarrow^* b_1 \dots b_n$

The length of this derivation is exactly the same as the length of the old derivation because the same rules are applied, but in a different order.

The length of the string $b_1 \dots b_i z b_{i+j+1} \dots b_n$ is $n - j + 1$. Applying the induction hypothesis we know now that the length of the derivation $s \Rightarrow^* b_1 \dots b_i z b_{i+j+1} \dots b_n$ is at most $(2|P'| + 1) \times ((n - j) + 1)$.

The length of the derivation $b_1 \dots b_i z_1 \dots z_j b_{i+j+1} \dots b_n \Rightarrow^* b_1 \dots b_n$ is maximally $j|P'|$.

$ld(G', n) \leq$
$((2|P'| + 1)((n - j) + 1)) + 1 + j|P'| =$
$(2|P'| + 1)n - 2j|P'| - j + (2|P'| + 1) + 1 + j|P'| =$
$(2|P'| + 1)n + 2|P'| + 2 - j|P'| - j =$
$(2|P'| + 1)n + (2 - j)(|P'| + 1)$

because $j \geq 2$, also $ld(G', n) \leq (2|P'| + 1)n$ □.

Take an arbitrary acyclic *context-sensitive* grammar.

4.4.2. LEMMA. $ld(G, n) \leq (2|P| + 1)n$.

Proof: Every derivation in an acyclic context-sensitive grammar is a derivation in the context-free backbone. The number of rules in the context-free backbone is at most the number of rules in the acyclic context-sensitive grammar. Two context-sensitive rules can have the same context-free backbone. But if $|P'| < |P|$ and $ld(G, n) \leq (2|P'| + 1)n$ then also $ld(G, n) \leq (2|P| + 1)n$. □

4.4.3. THEOREM. *The problem UNIFORM RECOGNITION FOR ACYCLIC CONTEXT-SENSITIVE GRAMMAR is in NP.*

Proof: A nondeterministic algorithm can guess every (bottom-up) replacement of some substring until the startsymbol has been found. This process will not take more steps than the length of the longest derivation. The longest derivation in an acyclic context-sensitive grammar has linear length. Therefore, this nondeterministic algorithm runs in linear time and it recognizes exactly $L(G)$. □

4.4.4. THEOREM. *There is a transformation f of 3-SAT to UNIFORM RECOGNITION FOR ACYCLIC CONTEXT-SENSITIVE GRAMMAR.*

Proof: First we transform the instances of 3-SAT to those of UNIFORM RECOGNITION FOR ACYCLIC CONTEXT-SENSITIVE GRAMMAR. An example of this transformation is:

$(\neg\, u_3 \vee u_2 \vee \neg\, u_1) \wedge (u_3 \vee \neg\, u_2 \vee u_1)$, a 3-SAT instance, is transformed into
"ini $\neg\, u_3\ u_2\ \neg\, u_1\ u_3\ \neg\, u_2\ u_1$".

The transformation should be done as follows.

The symbols "$\vee$", "$\wedge$" and the brackets "(" and ")" are left out of the new formula in order to keep the grammar smaller. An extra symbol is added in front of the formula. This symbol has to initialize all variables. We use the symbol "ini" for it and we call it the ini-symbol.

In Appendix B.1 the grammars for all different m (the number of variables in the formula) can be found. The rules of the form $A \to [B]C$ should be read as $B_1A \to B_1C$ (the B is the context). The terminal symbols are: $\Sigma = \{\text{ini}, \neg, u_i\}$ $(1 \leq i \leq m)$. The startsymbol S is "s". The number of rules of the grammar is cubic in m. We can show how this grammar recognizes a satisfiable formula of 3-SAT by applying the grammar rules bottom-up.

All u_i are initialized as true or false and their values are sent through the formula from left to right. Most nonterminal symbols have three subparts: the original terminal symbol, some variable and the value of that variable (true or false). E.g. the symbol "u_3u_2t" has been derived (bottom-up) from the terminal symbol u_3 and contains the information that u_2 has been made true.When the value of u_i crosses u_i, u_i

is turned into true or false (t or f). When e.g. u_3 "hears" from its left neighbour that u_3 has been initialized as false, "u_3u_2t" will be replaced by "fu_3f"[2].

We end up with the ini-symbol followed by a sequence of t's and f's. These sequences together form an "s" when none of the clusters of truthvalues contains three f's. The values of the u_i can only be sent in a fixed order: first u_1, then u_2 etc. When not all values are sent, the u's are not changed into t or f. For every variable we can send only one value. Hence only satisfiable formula's can form an "s". The grammar generates exactly all satisfiable formulas.□
Appendix B.2 contains an example of a derivation of the formula "ini u2 ¬ u3 u1" (where $m = 3$).

4.4.5. THEOREM. *f is polynomially computable.*

Proof: The transformation of instances is polynomial. The number of grammar rules is cubic in m, the number of variables. □

4.4.6. THEOREM. *The problem UNIFORM RECOGNITION FOR ACYCLIC CONTEXT-SENSITIVE GRAMMAR is NP-complete.*

Proof: Follows from Theorems 4.4.3, 4.4.4 and 4.4.5. □

4.5 Discussion

In the introduction we said that we wanted to introduce a formalism that allows parse trees with crossing branches, which has a simple definition, and which is computationally tractable. The last goal has been achieved only partially. The uniform recognition problem is NP-complete. If the grammar is fixed, then the recognition problem is in P. An aside here: the complexity is $\mathcal{O}(n^k)$ where k depends on the grammar. There is no fixed k.

An interesting question is which of the two problems is more relevant: the uniform recognition problem or the fixed grammar recognition problem. Barton Jr., Berwick and Ristad (1987) argue that the fixed grammar problem is not interesting because it is about languages, and not about grammars. The definitions they use are the following. The universal recognition problem is:

> Given a grammar G (in some grammatical framework) and a string x, is x in the language generated by G?

This problem is contrasted with the fixed language recognition problem:

> Given a string x, is x in some independently specified set of strings L?

[2] "¬u_3f u_3u_2t" will be replaced by "tu_3f": ¬ u_3 must get the value *true* when u_3 is initialized as *false*.

If we use the notation of Garey and Johnson (1979), we see that there are in fact not two but three different ways to specify recognition problems. Suppose we have a definition of context-sensitive grammars and the languages they generate. Then we can define the universal or uniform recognition problem as follows.

UNIFORM RECOGNITION FOR CONTEXT-SENSITIVE GRAMMARS
INSTANCE: A context-sensitive grammar $G = (V, \Sigma, S, P)$ and a string $w \in \Sigma^*$.
QUESTION: Is w in the language generated by G ?

There are two forms of the fixed language problem. Suppose we have a grammar ABCGRAM that generates the language $\{a^n b^n c^n | n \geq 1\}$. The following two problems can be defined.

RECOGNITION FOR CONTEXT-SENSITIVE GRAMMAR ABCGRAM
INSTANCE: A string $w \in \Sigma^*$.
QUESTION: Is w in the language generated by G ?

MEMBERSHIP IN $\{a^n b^n c^n | n \geq 1\}$
INSTANCE: A string $w \in \Sigma^*$.
QUESTION: Is w in $\{a^n b^n c^n | n \geq 1\}$?

The complexity of RECOGNITION FOR CONTEXT-SENSITIVE GRAMMAR ABCGRAM is identical to the complexity of MEMBERSHIP IN $\{a^n b^n c^n | n \geq 1\}$ because any algorithm that solves the first problem solves the second too and vice versa. The difference lies in the way the problems are specified. Barton Jr. et al. (1987) only consider the type of problems where languages are specified in some other way than by giving a grammar that generates the language. They argue that this type of problems is not interesting because the grammar has disappeared from the problem, and that it is more interesting to talk about families of languages/grammars than about just one language. However, we see that the grammar does not necessarily disappear in the fixed language recognition problem. If the grammar is part of the specification, we can talk about the fixed language problem in the context of families of grammars. We do this by abstracting over the fixed grammar. E.g., we can try to prove that *for every G*, RECOGNITION FOR CONTEXT-SENSITIVE GRAMMAR G is PSPACE-complete. Abstracting from the grammar is different from putting the grammar as input on the tape of a Turing machine, as is done in the uniform recognition problem.

If we want to answer the question which of the two problems is more relevant for us, we first have to answer the question: "Relevant for what?". Complexity analysis of grammatical formalisms serves at least two purposes. First, it helps us to design algorithms that can deal with natural language efficiently. Secondly, it can judge grammar formalisms on their psychological relevance.

It is hard to say whether the uniform problem or the fixed grammar problem is more relevant to efficient sentence processing. An argument in favor of the uniform recognition problem is the following. Usually, grammars are very big, bigger than

the sentence length. They have a strong influence on the runtime of an algorithm in practice. Therefore we can not simply forget the grammar size as is done in the fixed language recognition problem. On the other hand, we can argue in favor of the fixed language problem as follows. In many practical applications the task that an algorithm has to fullfill is the processing of sentences of some given language (e.g. a spelling checker for English). Practical algorithms have to decide over and over whether strings are in the language generated by some fixed grammar. Only in the development phase of the application the grammar changes. This argues for the relevance of the fixed grammar problem. It is not clear which of the two problems is more relevant in the design of efficient algorithms.

The psychological relevance is a very hard subject (shortly discussed in Barton Jr. et al. (1987, pp. 29,74)). Humans can process natural language utterances fast (linear in the length of the sentence). It seems that we have to compare this with the complexity of the fixed language problem. We follow the same line of reasoning more or less that was used in the previous paragraph. The problem that is solved by humans is the problem of understanding sentences of one particular language (or two, or three). People can not change their "built-in grammar" at will. Barton Jr. et al. (1987) remark that natural languages must be acquired, and that in language acquisition the grammar changes over time. This change is a very slow change, however, compared to the time needed for sentence processing.

Our conclusion is that both the uniform and the fixed language problem are interesting. Which of the two is more important depends on what perspective one takes.

Part II

Programming in Logic

Chapter 5

Complexity of Prolog Programs

5.1 Introduction

This chapter is about the complexity of Prolog programs. The worst case time complexity of programs written in an imperative language (like Pascal or C) can be estimated by straightforward means. These programs are deterministic so we can follow the execution step by step. The number of steps is counted by estimating the cost of smaller procedures, e.g. multiplying the number of times that a "while" loop is executed with the number of steps needed in the loop. A disadvantage is that the code of larger programs gets incomprehensible very soon. This is solved by presenting pseudo-code. Howver, in pseudo-code the reader has to guess the details. In fields like computational linguistics and artificial intelligence we often see algorithms explained by "real" Prolog code. This can be done because real Prolog code is easier to read than, e.g., C or Pascal code. An algorithm presented this way, has no open-ended details. For Prolog programs, however, the complexity analysis is not so easy. The main problem is that Prolog programs are intended to be non-deterministic. Computers are deterministic machines, however. Therefore any Prolog interpreter has to deal with the non-determinism in some deterministic way. Standard interpreters perform a depth-first search through the search space. In case of a choice point the interpreter takes a decision. If that decision appears to be wrong later, the interpreter reverses the decision and tries another possibility. This mechanism is called backtracking. Backtracking is what makes the analysis of Prolog programs so hard. The only attempt to estimate the runtime of Prolog programs in the context of standard interpreters is from Lin (1993). This method is discussed in the next section.

This chapter does not solve the problem of estimating the runtime for standard interpreters. However, we can estimate the time complexity easier than in Lin (1993) if we use an interpreter called the *Earley interpreter*. The Earley interpreter does not backtrack, but it keeps an administration of what goals have been tried and what the result was. It differs in two ways from the standard interpreter:

- Improved proof search. Prolog programs have a very clear meaning from a logical point of view. Standard interpreters do not behave properly however. They do not always find a proof although there exists one, because they end up in an infinite loop. One can "program around" this but then we leave the path of declarative programming. The Earley interpreter does what it should do. It can only get in a loop if terms can grow arbitrarily long. The method presented in this chapter is meant to be used for problems with a finite search space, i.e., decidable problems.
- Longer runtime. Because the interpreter has to do a lot of bookkeeping the runtime will be longer in general. This is the major disadvantage of the method presented here: in order to estimate the runtime we use an interpreter that increases the runtime. Lin (1993) does not have this disadvantage. There are two arguments in favor of our method. First, there are cases in which the bookkeeping can speed up algorithms as well. It can even speed up an exponential time algorithm to a polynomial time algorithm. Second, the overhead is small. The overhead is linear in the *size of the terms* during execution of the Prolog program. When the size of the terms does not grow exponentially but remains polynomial, we stay in the same complexity class (for classes that are insensitive for the degree of the polynomial, like $\mathcal{P}$). The method presented in this chapter is suited especially to prove that some problem is in $\mathcal{P}$.

The main reason to switch from the standard interpreter to the Earley interpreter is the possibility to prove runtime bounds for Earley interpreters in a pretty straightforward way. We will describe a simple method to deduce the runtime of an algorithm from two sources: the number of possible instantiations of the variables in the program and the length of the instantiations. If a Prolog programmer knows how the variables in his program behave, he can deduce the runtime in a simple manner.

The main idea behind our approach is the following. The Earley interpreter stores all attempts to prove some goal (i.e. it stores all "procedure" calls). Furthermore it stores all solutions (and all partial solutions). Because of this we can be sure that every procedure is executed only once for every call. When the procedure is called a second time the answer can be looked up and this costs only very little time. This is called *memoization*, *tabulation* or *tabling*. The search strategy is called Earley Deduction (or OLDT resolution). The Earley Deduction proof procedure is due to Warren (1975). It was first published in Pereira and Warren (1983). A good introduction is Pereira and Shieber (1987, pp. 196-210). Similar ideas can be found in Warren (1992), Tamaki and Sato (1986) (OLDT resolution), Vieille (1989) (SLD-AL resolution) and van Noord (1993).

The fact that all problems are solved only once makes it much easier to estimate the time complexity: we only have to count the number of procedure calls multiplied with the amount of time spent in the procedure for each call.

The structure of this chapter will be as follows. First we describe what other research has been done on this topic. Then we give a short introduction to Prolog. We describe the language and show its logical (or declarative) semantics. After that, we describe a non-deterministic interpreter that does exactly what should be done according to the declarative meaning. Then we show two methods to make the interpreters

deterministic. The first one leads to the standard interpreter. The second method leads to the Earley interpreter.

When it is clear how the Earley interpreter works we start our complexity analysis. The result of the counting will be a complexity formula in which one has to fill in the length and the number of all possible instantiations for the variables. The estimate we obtain is rather pessimistic. In the next chapter a more efficient Earley interpreter is presented. With this interpreter we obtain much better complexity formulas. In this chapter we present these formulas (and we postpone the way how they are obtained). After this we give some examples. We will sketch some ideas about further research. Finally we will say something about existing implementations of Prolog interpreters which are able to follow the search strategy we describe here.

5.2 Other Research

The complexity of Prolog programs has remained largely unexplored up till now. The PhD thesis of Lin (1993) and the article of Debray and Lin (n.d.) are the only treatments of the problem of complexity of Prolog programs. Lin (1993) describes some related work. In this section we will explain the ideas behind the approach of Lin and Debray and compare it to our approach.

Lin (1993) starts from the work that has been done on functional programming. Contrary to logic programs, functional programs are deterministic. The functional programming approach has been extended in order to deal with non-determinism. The second starting point is that the analysis is done (semi-)automatically. A system called CASLOG (Complexity Analysis System for LOGic) has been implemented. This system takes as input an annotated Prolog program and produces formulas which are estimates for the time and space complexity.

These formulas are computed by looking at *size differences*. In imperative programming languages iteration is implemented with *while statements*. Examination of the *guard* of the while statement tells how many times the loop is executed. In functional and logic programming, iteration is implemented by *recursion*. The depth of a recursion can not be estimated as easily as the number of iterations in a while statement. The depth of a recursion can be estimated by looking at the size differences of the terms in the procedure call. Suppose we call a procedure with a list of length n. Within this procedure, there is a recursive call with a list of length $n-1$. Then we can conclude that the depth of the recursion is n. This description is only a clue for the method that CASLOG uses. We refer to Lin (1993) for more details.

Our approach differs from Lin's as follows. Basically, our method is by hand and Lin's is automatically. The method of Lin is fairly complex. When the system computes some estimate it is hard to see *why* the estimate is correct. In our approach it is clear how the estimates are obtained (because the analysis is done by hand). A second difference is that Lin's approach does not always work well. Take for example a program that computes the transitive closure of a graph. The difference in the procedure calls is not in the size of the arguments, the arguments are just vertices. The advantage of doing the analysis by hand is that we can apply the method to

problems where the size difference is not important. The third difference is that Lin uses the standard proof search (depth-first with backtracking) whereas we use OLDT-resolution (or Earley Deduction). The standard proof search is much more common, OLDT resolution is hardly used in practice as yet. For experienced Prolog programmers, the main drawback of OLDT resolution is that it is slower than standard search in general. The advantage is that it has a better semantics, i.e., there is no gap between procedural and declarative semantics.

A similarity between the two approaches is that both methods are restricted to *well-moded* programs.With *modes* we express whether an argument of a predicate behaves as input or as output. Well-modedness is a syntactic restriction on occurrences of variables. (this is explained further in section 5.6). It would be an interesting project to see which elements of the approach by Lin can be incorporated in our approach.

5.3 Prolog

5.3.1 Introduction

Prolog stands for programming in logic. The logic that is used is the *Horn fragment of first order logic*. A good introduction is Lloyd (1987). Logic has a syntactic component; the proof theory, and a semantic component; the model theory. In this thesis we will focus on the proof-theoretic side. Lloyd introduces logic programming from both perspectives.

Conventional programming languages are procedurally oriented. Prolog introduces a declarative view on programming. One has to give a description of the problem one wants to solve. Once this description has been provided an interpreter will solve the problem. Ideally, the programmer should not bother about the way the interpreter solves the problem. This is the procedural part of the problem and it should be "hidden". However, in real life it is not always possible to separate the description of the problem and the search for solutions.

The reason for this is that that the declarative meaning and the procedural meaning of Prolog programs often differ. A Prolog program consists of a set of axioms (or unit clauses) and a set of rules. The declarative meaning is defined as follows. Some expression is true if it can be derived from the axioms via the rules. When the expression is not derivable it is false. The declarative meaning is what the interpreter *should* do. The procedural meaning is what the interpreter in fact does. That is, some goal is true if the interpreter *can find* a derivation, and it is false if it *can not find* a derivation (and terminates).

If we want to implement an Earley interpreter we face two choices. We have to decide whether we want to follow a breadth-first strategy or a depth-first strategy. Furthermore, we can choose between an exhaustive search, that generates all solutions, or a non-exhaustive search, which generates only one solution. Any interpreter that performs a breadth-first non-exhaustive search is *complete*, i.e., it will find a proof if there is one.

The Earley interpreter has more advantages than the improved proof search. It

is much easier to give estimates for the complexity of problems. The solution of a problem is given by a declarative description plus some interpreter that finds proofs. The complexity of the problem is the number of steps that the interpreter takes in finding proofs (or in finding out that there are no proofs).

5.3.2 Definitions

In this section we will give definitions of the syntax and the declarative (non-procedural) meaning of programs. In the second half of the section we will define the procedural meaning.

Syntax. The language has a countable infinite set of *variables*, and countable sets of *function* and *predicate symbols*. Each function symbol f and each predicate symbol p is associated with a natural number n, the *arity* of the symbol. A function symbol with arity 0 is referred to as a constant. A *term* in the language is a variable, a constant, or a *compound term* $f(t_1, \ldots, t_n)$, where f is a function symbol with arity n, and t_1 through t_n are terms. A term is *ground* if it does not contain a variable.

We use the convention that Prolog variables are written with an uppercase letter. Constants, predicate and function symbols in Prolog start with a lowercase letter. Terms represent complex data structures. For instance, the date "May 1 1993" can be represented as the term `date(1,may,1993)`. Any date in that same month can be represented as the term `date(Day,may,1993)`.

An *atom* $p(t_1, \ldots, t_n)$ consists of an n-ary predicate symbol p and n terms t_i as the arguments of p. An atom is *ground* if all its arguments are ground terms. There are three types of *Horn clauses*: *facts, rules, and queries*.

- Rules declare atoms that are true depending on some conditions. They are written as 'p :- $q_1, \ldots, q_n$.', where p, q_i are atoms. Read this as: p holds if q_1 and ...and q_n. The atom p is called the *head*, and the sequence of atoms $q_1, \ldots, q_n$ is the body.
- Facts declare atoms that are unconditionally true, i.e. they are rules with an empty body. They are written as 'p.', where p is an atom.
- By means of queries the program can be asked what atoms are true. A query is of the form '?- $q_1, \ldots, q_n$.', where all q_i are atoms.

A *definite clause* (or *program clause*, or just *clause*) is a rule or a fact. A *predicate definition* consists of a finite number of definite clauses, all with the same predicate symbol in the head. A *logic program* (or *program*) consists of a finite number of predicate definitions.

When we try to prove an atom, we sometimes say that we try to prove a *goal*. A function symbol is also called a *functor*. A *substitution* is a mapping from variables to terms that is the identity mapping at all but finitely many points. With $Var(X)$ we denote the set of variables occurring in X.

Declarative meaning. With a definition of facts and rules we can define truth values for queries. This is called the declarative meaning of Prolog programs.

5.3.1. DEFINITION. A query $?- Q_1, Q_2, \ldots, Q_n$ is *true* if and only if there is a substitution σ such that the atoms $\sigma(Q_1)$ and ... and $\sigma(Q_n)$ are true.

An atom G is *true* if and only if there is a clause $C :- B_1, B_2, \ldots, B_n$ in the program ($B_1, \ldots, B_n$ can be empty, then C is a fact) and a substitution τ such that $\tau(C)$ is identical to G and the atoms $\tau(B_1)$ and ... and $\tau(B_n)$ are true.

If one of the $\sigma(Q_i)$ is not ground, then the amount of solutions is infinite immediately. Every solution where the variable in $\sigma(Q_i)$ is replaced by some other term, also makes the query true. These substitutions are less interesting because they can all be represented by one single, more general, substitution. Therefore, we are interested in the *most general* substitutions. A substitution σ_1 is more general than a substitution σ_2 if there is a substitution σ_3, not the identity, such that $\sigma_2 = \sigma_3 \circ \sigma_1$, where $\circ$ is the function composition operator. A substitution σ is a most general substitution for a query if the substitution makes the query true and there is no more general substitution that makes the query true. There may be more most general substitutions (even infinitely many).

Procedural meaning. If we want to build an interpreter, Definition 5.3.1 is not very helpfull. We have to guess substitutions and program clauses in order to find a proof. Guessing the substitutions can be eleminated by using *most general unifiers*.

We say that a term Z is a *unifier* for the terms X and Y if there is a substitution σ such that $Z = \sigma(X)$ and $Z = \sigma(Y)$. We say that term X is *more general* than term Y if there is a substitution σ' such that $Y = \sigma'(X)$ and $Y \neq X$. A term Z is the *most general unifier* of X and Y if it is a unifier of X and Y and there is no unifier that is more general. The most general unifier is unique modulo variable renaming. We assume that $Var(mgu(A,B)) \cap Var(A) = \emptyset$ and $Var(mgu(A,B)) \cap Var(B) = \emptyset$ for all A, B (every most general unifier contains fresh variables) (we standardize the variables apart). Two terms T_1 and T_2 are *alphabetic variants*, if there are substitutions σ_1 and σ_2 such that $T_1 = \sigma_1(T_2)$ and $T_2 = \sigma_2(T_1)$. All these notions can be defined on atoms instead of terms in the same way. There exist unification algorithms that are linear in the size of the terms (Paterson and Wegman 1978).

A non-deterministic interpreter, that can prove queries w.r.t. the semantics in Definition 5.3.1, is in Figure 5.1. The semantics is the same as the semantics in Lloyd (1987) (SLD-refutations under the left-most selection rule). The interpreter generates the most general substitutions only (often called the *computed answer substitutions*).

The only non-determinism left is the guessing of a clause. In standard interpreters this non-determinism is eliminated by performing a depth-first search. Every time the interpreter has to guess a clause it takes the first available. If it turns out later that this choice was wrong, i.e. that no proof can be found, we try the second possibility and so on.

This strategy often leads to problems. Consider the program `PATH`. It computes the reflexive transitive closure of a graph.

Prove list of goals:

Given a list of goals $Q_1, \ldots, Q_n (n > 1)$, prove Q_1. The result will be a substitution σ_1 such that $\sigma_1(Q_1)$ is derivable. Then prove the list of goals $\sigma_1(Q_2), \ldots, \sigma_1(Q_n)$. The result of proving $\sigma_1(Q_2), ..., \sigma_1(Q_n)$ is a substitution $\sigma_n \circ \ldots \circ \sigma_2$. The result of proving $Q_1, \ldots, Q_n$ is $\sigma_n \circ \ldots \circ \sigma_2 \circ \sigma_1$.

Prove goal:

Given a goal G, guess a clause H `:-` $B_1, B_2, \ldots, B_n$ in the program and compute $mgu(G, H)$. Suppose $mgu(G, H) = f(H) = g(G)$. Prove the list of goals $f(B_1), f(B_2), \ldots f(B_n)$. The result is the substitution h. The result of proving goal G is the substitution $h \circ g$.

Figure 5.1: Non-deterministic interpreter

```
% Sample program PATH

path(X,Z) :-
        path(X,Y),
        edge(Y,Z).
path(X,X).

edge(a,b).
edge(b,c).
edge(c,a).
edge(c,d).
```

Suppose we have a query `?- path(a,d).` A depth-first searching Prolog interpreter tries to prove the following goals: `path(a,d)`, `path(a,Var1)`, `path(a,Var2)`, `path(a,Var3)`, etc. The interpreter never gets out of this loop.

We now introduce the Earley interpreter. It is defined as follows. We define a meta-interpreter, an interpreter written in Prolog. The meta-interpreter is almost deterministic (with the *if-then-else* constructor we could make it deterministic). The interpreter in Figure 5.1 (with some extensions for the impure parts of the meta-interpreter) can be used to interpret the code of the meta-interpreter. We introduce the infix predicate symbol `::-`. Clauses of the program are written as `Head ::- Body`. This prevents that the meta-interpreter executes its own code. The basic datastructure we use is called the *item*. Items have the same form as clauses: they are pairs of heads and bodies. The head of the item is the head of some clause after some substitution. The body of the item is a (possibly empty) remainder of the same clause after the same substitution. Items are used to store partial results. This is done as follows. Consider the interpreter in Figure 5.1. We have to prove some goal, and therefore we take an arbitrary clause from the program. After computing the *mgu* of the goal and the head of the clause, we obtain the item $\langle f(C), [f(B_1), f(B_2), \ldots, f(B_n)] \rangle$. Now we try to prove $f(B_1)$.

This gives us the substitution σ_1 and the new item $\langle \sigma_1(f(C)), [\sigma_1(f(B_2)), \ldots, \sigma_1(f(B_n))] \rangle$. We prove $\sigma_1(f(B_2))$, find σ_2, and obtain $\langle \sigma_2(\sigma_1(f(C))), [\sigma_2(\sigma_1(f(B_3))), \ldots, \sigma_2(\sigma_1(f(B_n)))] \rangle$. In every step the body becomes shorter. Finally, the body is empty. The final item $\langle \sigma_n(\ldots(\sigma_1(f(C)))), [\,] \rangle$ is a solution for the goal we tried to prove.

The data structure the Earley interpreter uses has been described now. The control structure is as follows. We keep an *agenda* of items that wait to be processed and a *table* of items that have been processed. We make sure that all items in the table are alphabetically different. Two items that are alphabetic variants are considered as being equal. Processing an item from the agenda is done as follows. We first look whether the item (or an alphabetic variant) occurs in the table. If it does, we can simply discard it because it has been processed earlier. If it does not occur in the table (there is no alphabetic variant in the table) there are two possibilities.

- The body is empty. That means that the item is a solution. We combine the solution with the items in the table that are waiting for that solution. This gives us new items which are placed on the agenda again. This operation is called completion.
- The body is not empty. Two operations are executed:
  - prediction. The first element of the body is unified with heads of clauses in the program. New items are put on the agenda again.
  - completion. The first element of the body is combined with solutions in the table.

We will implement two interpreters: one that generates all solutions and one that stops after it has found the first solution. If we want to generate all solutions, then we stop when the agenda is empty. When we want only one solution, we can stop when the first solution of the (top) query appears in the agenda.

The algorithms sketched above can be implemented in Prolog as follows. The main predicate for the one-solution interpreter is `prove_one`. It is called with the goal we want to prove as argument. For the all-solutions interpreter the main predicate is `prove_all`. This predicate is called with the goal and with a variable that will be instantiated to the list of all solutions. We can change between depth-first and breadth-first behaviour simply by swapping two arguments in some `append` predicate.

The program clauses are of the form `Goal :- Body`, where Body is a (possibly empty) list of atoms, in order to separate them from the clauses of the interpreter. The program is given as a whole first and in little parts with comment later.

```
% Earley interpreter.
% needs: findall, member, append, numbervars.

?- op(1150,xfx,::-).

prove_all(Goal,Solutions) :-
        findall(item(Goal,Goals),(Goal ::- Goals),Agenda),
        extend_items_all(Agenda,[],Table),
        findall(Goal,member(item(Goal,[]),Table),Solutions).

extend_items_all([],Table,Table).
extend_items_all([Item|Agenda1],Table1,Table2) :-
        memberv(Item,Table1),
        extend_items_all(Agenda1,Table1,Table2).
extend_items_all([Item|Agenda1],Table1,Table3) :-
        \+ memberv(Item,Table1),
        Table2 = [Item|Table1],
        new_items(Item,Table1,Items),
        append(Items,Agenda1,Agenda2),  % depth-first search
%       append(Agenda1,Items,Agenda2),  % breadth-first search
        extend_items_all(Agenda2,Table2,Table3).
```

```
prove_one(Goal,YN) :-
        findall(item(Goal,Goals),(Goal ::- Goals),Agenda),
        extend_items_one(Agenda,[],Goal,YN).

extend_items_one(Agenda,Table,Goal,yes) :-
        member(item(Goal,[]),Agenda).
extend_items_one([],Table,Goal,no) :-
        \+ member(item(Goal,[]),Table).
extend_items_one([Item|Agenda1],Table,Goal,YN) :-
        \+ member(item(Goal,[]),Table),
        memberv(Item,Table),
        extend_items_one(Agenda1,Table,Goal,YN).
extend_items_one([Item|Agenda1],Table1,Goal,YN) :-
        \+ member(item(Goal,[]),Table1),
        \+ memberv(Item,Table1),
        Table2 = [Item|Table1],
        new_items(Item,Table1,Items),
%       append(Items,Agenda1,Agenda2),  % depth-first search
        append(Agenda1,Items,Agenda2),  % breadth-first search
        extend_items_one(Agenda2,Table2,Goal,YN).
```

```
new_items(item(Goal1,[]),Table,Items) :-
        findall(item(Goal2,Goals),
                member(item(Goal2,[Goal1|Goals]),Table),
                Items).
new_items(item(Goal1,[Goal2|Goals1]),Table,Items) :-
        findall(item(Goal2,Goals2),(Goal2 ::- Goals2),Items1),
        findall(item(Goal1,Goals1),
                member(item(Goal2,[]),Table),
                Items2),
        append(Items1,Items2,Items).

memberv(Item,Table) :-
        member(Item2,Table),
        variant(Item,Item2).

variant(X,Y) :-
        \+ (\+ (numbervars(X,0,_),numbervars(Y,0,_),X == Y)).
```

The definition of the predicates `append`, `member`, `\+` (not), `findall` and `numbervars` follows the standard conventions. The predicate `append` is a predicate for the concatenation of two lists. The predicate `member` is a predicate for membership of a list. When `findall(X,condition(X),Solutions)` has been proved, `Solutions` = $\{X \mid condition(X)\}$. The predicate `numbervars` replaces all variables in a term by special constants. This operation makes two terms identical when they are alphabetic variants.

A sample program is (facts are represented as rules with an empty body):

```
% Sample program PATH

path(X,Z) ::-
       [path(X,Y),
        edge(Y,Z)].
path(X,X) ::- [].

edge(a,b) ::- [].
edge(b,c) ::- [].
edge(c,a) ::- [].
edge(c,d) ::- [].
```

The predicates just given are repeated here with a little comment.

```
prove_all(Goal,Solutions) :-
        findall(item(Goal,Goals),(Goal ::- Goals),Agenda),
        extend_items_all(Agenda,[],Table),
        findall(Goal,member(item(Goal,[]),Table),Solutions).
```

The query (`Goal`) is used to predict items. These items are put in the agenda. The interpreter is started with `extend_items_all`. When `extend_items_all` is finished

we search in the table for all solutions.

```
extend_items_all([],Table,Table).
```

If the agenda is empty we are finished.

```
extend_items_all([Item|Agenda1],Table1,Table2) :-
        memberv(Item,Table1),
        extend_items_all(Agenda1,Table1,Table2).
```

If an item from the agenda is in the table, it can be discarded. Observe that we test whether there is an *alphabetic variant*. This is crucial. If we did not do this and would consider all goals that are alphabetic variants as being different, the interpreter would loop on the PATH program just like the depth-first interpreter.

```
extend_items_all([Item|Agenda1],Table1,Table3) :-
        \+ memberv(Item,Table1),
        Table2 = [Item|Table1],
        new_items(Item,Table1,Items),
        append(Items,Agenda1,Agenda2), % depth-first
%       append(Agenda1,Items,Agenda2), % breadth-first
        extend_items_all(Agenda2,Table2,Table3).
```

If an item is not in the table as yet, it is added and new items are generated. These new items are put in front or at the back of the agenda, corresponding with depth-first and breadth-first behaviour respectively.

```
new_items(item(Goal1,[]),Table,Items) :-
        findall(item(Goal2,Goals),
                member(item(Goal2,[Goal1|Goals]),Table),
                Items).
```

`item(Goal1,[])` is a solution. It is combined with items in the table that wait for that solution: `item(Goal2,[Goal1|Goals])`. Because `Goal1` has been proved we create the item `item(Goal2,Goals)`. This is called completion.

```
new_items(item(Goal1,[Goal2|Goals1]),Table,Items) :-
        findall(item(Goal2,Goals2),(Goal2 ::- Goals2),Items1),
        findall(item(Goal1,Goals1),
                member(item(Goal2,[]),Table),
                Items2),
        append(Items1,Items2,Items).
```

The item we have to process `item(Goal1,[Goal2|Goals1])` is not a solution, but it is waiting for a proof of `Goal2`. The first `findall` is the prediction step: `Goal2` is unified with heads of clauses in the program (if an item with `Goal2` as the first element in the body has been processed earlier, then we will predict items that are already in the table). The second `findall` searches the table for solutions. This is the "mirror image" of the completion step in the previous clause.

```
memberv(Item,Table) :-
        member(Item2,Table),
        variant(Item,Item2).

variant(X,Y) :-
        \+ (\+ (numbervars(X,0,_),numbervars(Y,0,_),X == Y)).
```

Here we check whether there is an alphabetic variant in the table. The predicate `findall` is defined in such a way that two items never share any variables. *Variables are only shared within an item.*

The one-solution interpreter only differs from the all-solutions interpreter in the fact that the first stops when the first solution is found, whereas the latter goes on until the agenda is empty.

The one-solution interpreter has two advantages over the standard Prolog interpreter with a depth first strategy. The first is that the declarative meaning and the procedural meaning coincide: the interpreter answers "yes" to a query if and only if the query is indeed derivable from the facts. Standard interpreters get in an infinite loop in our example program PATH. The second advantage is that "a problem is never solved twice". The reuse of results of subcomputations in Prolog interpreters is called *memoing* or *tabling*. The technique is also known in general as *dynamic programming*. It can be seen that solutions are never computed twice as follows. Suppose we have two items `(Head1,[Goal|Tail1])` and `(Head2,[Goal|Tail2])`. The predictor will generate the same set of items (of the form `(Goal,Body)`) for both items. The first set of items will be processed normally. The elements of the second set will all be discarded when we try to move them from the agenda to the table because there are already duplicates in the table. The result is that the goal `Goal` is proved only once.

Memoing can save us a lot of time. Consider the program in Figure 5.2. It can be made arbitrarily long by adding `x4`, `x5`, etc.

```
s    :- x1a,c.
s    :- x1b.
x1a  :- x2a.
x1a  :- x2b.
x1b  :- x2a.
x1b  :- x2b.
x2a  :- x3a.
x2a  :- x3b.
x2b  :- x3a.
x2b  :- x3b.
x3a.
x3b.
```

Figure 5.2: Exponential versus quadratic.

When we use standard proof search the time needed to detect that there is no proof is exponential in the size of the program. When we use Earley Deduction the time

needed is linear in the length of the program (this follows from the complexity estimate that will be shown later).

5.4 Space Complexity

Because it is much easier to reason about space complexity than about time complexity of the interpreter that we gave in the previous section, we start with a space complexity analysis. This analysis is also a good stepping stone towards the time complexity analysis in the next section. We will define two ways to look at decidability problems in Prolog theorem proving. The first is:

PROLOG THEOREM PROVING
INSTANCE: A query Q and a program P.
QUESTION: Is there a substitution σ such that $\sigma(Q)$ follows from P?

This problem is undecidable in general. E.g., in (Shapiro 1984) it is described how a Prolog program can simulate a Turing Machine. The problem is semi-decidable: if the answer is yes we can give an algorithm that finds a proof in finitely many steps. An algorithm with this property is the one-solution interpreter. This interpreter searches its proofs under a breadth-first regime and is therefore guaranteed to stop if there exists a proof.

In this thesis we will define *classes of programs* as follows. We define predicates to be *fixed* or *free*. All programs in a class must have identical clauses for the fixed predicates, but differ in the clauses for the free predicates. In this thesis all free predicates are defined with ground facts only. When there are no free predicates, the class contains only one program. The set of clauses for the fixed predicates in some class is called the *core program*.

PROLOG THEOREM PROVING FOR CORE PROGRAM P
INSTANCE: A query Q and a set of clauses R.
QUESTION: Is there a substitution σ such that $\sigma(Q)$ follows from $P \cup R$?

P defines the fixed predicates, and R the free predicates. We define PROLOG THEOREM PROVING FOR PROGRAM P as a special case of PROLOG THEOREM PROVING FOR A CORE PROGRAM P, namely, when $R = \emptyset$. In the rest of this thesis we will only consider programs for which these problems are decidable, because we have nothing interesting to say about undecidable cases. The `path` program, that we presented earlier, consists of two predicates: `path` and `edge`. We can define the `path` predicate as fixed and the `edge` predicate as free. This enables us to give upperbounds for various graphs and not just for one graph (because we can vary the edges).

The space complexity of PROLOG THEOREM PROVING FOR CORE PROGRAM P is the size of the table when the computation of the one-solution interpreter has finished.

This is only true if all items in the agenda and the table are always different. We

modify the interpreter such that only items which neither occur in the agenda nor in the table are put on the agenda. This guarantees that, at any point in the computation, the size of the agenda plus the size of the table is smaller than the size of the final table. This modification can be implemented in Prolog as follows (for a breadth-first behaviour the new items must be appended at the back of the agenda):

```
extend_items_all([],Table,Table).
extend_items_all([Item|Agenda1],Table1,Table3) :-
        Table2 = [Item|Table1],
        new_items(Item,Table1,Items),
        add_items_df(Items,Agenda1,Table2,Agenda2),
        extend_items_all(Agenda2,Table2,Table3).

add_items_df([],Ag,_,Ag).
add_items_df([H|T],Ag,Table,Ag2) :-
        memberv(H,Ag),
        add_items_df(T,Ag,Table,Ag2).
add_items_df([H|T],Ag,Table,Ag2) :-
        memberv(H,Table),
        add_items_df(T,Ag,Table,Ag2).
add_items_df([H|T],Ag,Table,Ag2) :-
        \+ memberv(H,Ag),
        \+ memberv(H,Table),
        add_items_df(T,[H|Ag],Table,Ag2).
```

Because we can not predict when the first solution is found by the one-solution interpreter, we will assume the worst scenario, where the computation ends because the agenda is empty. In fact we estimate the complexity of the one-solution interpreter and the all-solutions interpreters simultaneously.

In order to estimate the size of the final table we have to count

- the number of items in the final table, and
- the length of these items.

We will index our clauses from now on. The function $l(i)$ denotes the number of atoms in the body of clause i. For every clause $C_i :- B_{i1}, B_{i2}, \ldots, B_{in}$ (i is the index, if i is fixed, we write n as a shorthand for $l(i)$), the items in the table are of the following form:

$\langle f(C_i), [f(B_{i1}), f(B_{i2}), \ldots, f(B_{in})]\rangle.$
$\langle \sigma_1(f(C_i)), [\sigma_1(f(B_{i2})), \ldots, \sigma_1(f(B_{in}))]\rangle.$
$\langle \sigma_2(\sigma_1(f(C_i))), [\sigma_2(\sigma_1(f(B_{i3}))), \ldots, \sigma_2(\sigma_1(f(B_{in})))]\rangle.$
$\vdots$
$\langle \sigma_n(\ldots(\sigma_1(f(C_i))), [\,]\rangle$

If we want to count the number of items, we have to estimate the number of possible substitutions f, $\sigma_1 \circ f$, $\sigma_2 \circ \sigma_1 \circ f$, $\ldots$, $\sigma_n \circ \ldots \circ f$, for every clause i in the program. In general the number of substitutions is infinite but we count here the number of

substitutions with a given query and program that occur when the proof procedure specified in the one-solution and all-solutions interpreters is executed. We count just the number of "alphabetically different" substitutions. We introduce a notation for the number of possible substitutions for a set of variables.

5.4.1. DEFINITION. The function $\#_i(S, j)$ (where S is a set of variables occurring in clause i of a program, $0 \leq j \leq l(i)$) returns the number of possible substitutions of the variables in S after proving $Bi1$ through Bij, i.e. the number of possible $\sigma_j \circ \ldots \circ \sigma_1 \circ f$ (we only count the substitutions for the variables in S) in the interpreter in Figure 5.1 for clause i. For $j = 0$, $\#_i(S, 0)$ is the number of substitutions f.

The j in the function denotes a position in the body between two atoms. The positions in the body and the various B_{ij} are related as follows:

$$\overset{0}{\bullet}\ B_{i1}\ \overset{1}{\bullet}\ \ldots\ldots\ B_{ij}\ \overset{j}{\bullet}\ B_{i(j+1)}\ \overset{j+1}{\bullet}\ \ldots\ldots\ B_{i(l(i))}\ \overset{l(i)}{\bullet}$$

The following example might clarify the definition of $\#_i$.

```
r :-
        p(D,E,F),
        q(D,E,F).
p(A,2,C).
p(1,B,C).
q(1,2,3).
q(1,2,4).
```

$\sigma_1 = \{D/D, E/2, F/F\}$ $\quad \sigma_2 \circ \sigma_1 = \{D/1, E/2, F/3\}$
$\sigma_2 = \{D/1, F/3\}$ $\quad \sigma_2' \circ \sigma_1 = \{D/1, E/2, F/4\}$
$\sigma_2' = \{D/1, F/4\}$ $\quad \sigma_2'' \circ \sigma_1' = \{D/1, E/2, F/3\}$
$\sigma_1' = \{D/1, E/E, F/F\}$ $\quad \sigma_2''' \circ \sigma_1' = \{D/1, E/2, F/4\}$
$\sigma_2'' = \{E/2, F/3\}$
$\sigma_2''' = \{E/2, F/4\}$

f is the identity substitution. The clause index of the first clause is 0. Now:
$\#_0(\{D, E, F\}, 1) = 2$: $\{D/D, E/2, F/F\}$ and $\{D/1, E/E, F/F\}$
$\#_0(\{D, E, F\}, 2) = 2$: $\{D/1, E/2, F/3\}$ and $\{D/1, E/2, F/4\}$
$\#_0(\{F\}, 1) = 1$: $\{F/F\}$
$\#_0(\{F\}, 2) = 2$: $\{F/3\}$ and $\{F/4\}$

When we are going to count the number of items, we first have to see which variables occur in the substitutions (i.e., for which variables the substitution is not the identity mapping). We will call these variables the *relevant variables*. Suppose we want to count the items of the following form:

$$\langle \sigma_j(\ldots\sigma_1(f(C_i))), [\sigma_j(\ldots\sigma_1(f(B_{i(j+1)}))), \ldots, \sigma_j(\ldots\sigma_1(f(B_{in})))] \rangle.$$

Then the following variables are *not* relevant:

- Variables that occur in $B_{i1}, \ldots, B_{ij}$, but neither in $B_{i(j+1)}, \ldots, B_{in}$ nor in C_i. These variables are not relevant, simply because they do not occur in the item.
- Variables that occur in $B_{i(j+1)}, \ldots, B_{in}$, but neither in $B_{i1}, \ldots, B_{ij}$ nor in C_i. These variables are not relevant, because they are uninstantiated.

The other variables are relevant. These can be divided in two groups:

- The variables in C_i. These are called the *head variables* $HV(i)$. $HV(i) = Var(C_i)$
- Variables that occur both in $B_{i1}, \ldots, B_{ij}$ and in $B_{i(j+1)}, \ldots, B_{in}$ but not in C_i. These variables are called the *relevant body variables* $RV(i,j)$. $RV(i,j) = (Var(B_{i1}) \cup \ldots \cup Var(B_{ij})) \cap (Var(B_{i(j+1)}) \cup \ldots \cup Var(B_{in}))$.

If we count the number of items in the final table, we obtain the following upper-bound:

$$\sum_{i=1}^{k} \sum_{j=0}^{l(i)} \#_i(HV(i) \cup RV(i,j), j)$$

We apply this formula in the example program PATH.

```
% Sample program PATH

path(X,Z) ::-
       [path(X,Y),
        edge(Y,Z)].
path(W,W) ::- [].

edge(a,b) ::- [].
edge(b,c) ::- [].
edge(c,a) ::- [].
edge(c,d) ::- [].
```

The number of items is:

$\#_0(\{X,Z\} \cup \emptyset, 0) + \#_0(\{X,Z\} \cup \{Y\}, 1) + \#_0(\{X,Z\} \cup \emptyset, 2) +$
$\#_1(\{W\} \cup \emptyset, 0) +$
$\#_2(\emptyset \cup \emptyset, 0) +$
$\#_3(\emptyset \cup \emptyset, 0) +$
$\#_4(\emptyset \cup \emptyset, 0) +$
$\#_5(\emptyset \cup \emptyset, 0)$

We know that the variables in this programs can only be substituted by a vertex in the graph (`a, b, c` or `d`). We denote the number of vertices in the graph as $|V|$ and the number of edges as $|E|$, and fill in the formula:

$$|V|^2 + |V|^3 + |V|^2 + |V| + 1 + \ldots + 1$$

Because $|E| \leq |V|^2$, the number of items in the final table is $\mathcal{O}(|V|^3)$.

The space complexity does not depend on the number of items only, but also on the length of the items. We introduce the function $\#\#_i$ that does not count the number of

substitutions, but the number of *symbols needed to write down all substitutions*. The number of instantiations and the length of the instantiations behave different if they are estimated for a set of variables. Suppose we have two variables, A and B. The number of possible substitutions for A is n_1. The number of possible substitutions for B is n_2. The average length of the substitutions for A and B is l_1 and l_2 respectively. Then the number of possible substitutions for $\#_i(\{A,B\},j)$ is $n_1 \times n_2$ (if A and B are independent from each other). The average length of all substitutions is $l_1 + l_2$, and $\#\#_i(\{A,B\},j)$ is $(n_1 \times n_2) \times (l_1 + l_2)$. We have to *multiply* the possibilities and *add* the lengths. Because variables are often dependent it will in general not be the case that $\#_i(\{A,B\},j) = \#_i(A,j) \times \#_i(B,j)$. We are not interested in constant factors, therefore we define that $\#_i(X,j) = \mathcal{O}(\#_i(X,j))$ and $\#\#_i(X,j) = \mathcal{O}(\#\#_i(X,j))$.

We assume that the length of clauses in the program is bounded by a constant. The space complexity of PROLOG THEOREM PROVING FOR CORE PROGRAM P is now:

$$\sum_{i=1}^{k} \sum_{j=0}^{l(i)} \#\#_i(HV(i) \cup RV(i,j), j)$$

In the example, we assume that the number of symbols needed to write down a vertex is bounded by a constant. In that case, the space complexity of the problem is $\mathcal{O}(|V|^3)$.

5.5 Time Complexity

In the previous section we saw how the space complexity of a problem can be estimated. In this section we will consider the time complexity. Observe that the Earley interpreter is deterministic. Therefore it can be converted to a program in an imperative language in a straightforward way. We will assume that this has been done. We will describe the time complexity of PROLOG THEOREM PROVING FOR CORE PROGRAM P in terms of this imperative interpreter. We count the number of steps of the all-solutions interpreter because we can not predict when the first solution of a problem has been found. Therefore we assume that the algorithm terminates when the agenda is empty.

We know that all items in the table and the agenda are different. Therefore the procedure `extend_items_all` will be executed as many times as there are items in the final table: $\sum_{i=1}^{k} \sum_{j=0}^{l(i)} \#_i(HV(i) \cup RV(i,j), j)$. Within this procedure, we have to execute the procedures `new_items` and `add_items_df`. Within the procedure `new_items` we perform *prediction* and *completion*. We can divide the table in two parts. First, we have items of the form `item(Goal1,[Goal2|Goals])`. The number of items of this form is $\sum_{i=1}^{k} \sum_{j=0}^{l(i)-1} \#_i(HV(i) \cup RV(i,j), j)$. Secondly, we have items of the form `item(Goal,[])`. There are $\sum_{i=1}^{k} \#_i(HV(i), l(i))$ such items in the final table (observe that $RV(i,l(i)) = \emptyset$). If we sum all completion steps, we see that every item of the form `item(Goal1,[Goal2|Goals])` is compared exactly once with every item of the form `item(Goal,[])` to check whether the second is a solution for the first. Therefore the

total number of completion steps is

$$\sum_{i=1}^{k} \sum_{j=0}^{l(i)-1} \#_i(HV(i) \cup RV(i,j), j) \times \sum_{i=1}^{k} \#_i(HV(i), l(i))$$

In a completion step, we have to unify two atoms. The time needed is the sum of the length of the two atoms. If the length of the atoms in the program is bounded by a constant, then the total time needed for all completion steps is (## is the sum of the length of all possible substitutions):

$$(\sum_{i=1}^{k} \sum_{j=0}^{l(i)-1} \#\#_i(HV(i) \cup RV(i,j), j) \times \sum_{i=1}^{k} \#_i(HV(i), l(i))) \; +$$

$$(\sum_{i=1}^{k} \sum_{j=0}^{l(i)-1} \#_i(HV(i) \cup RV(i,j), j) \times \sum_{i=1}^{k} \#\#_i(HV(i), l(i)))$$

The number of prediction steps is estimated as follows. We do a prediction step once for every item of the form `item(Goal1,[Goal2|Goals])`, thus $\sum_{i=1}^{k} \sum_{j=0}^{l(i)-1} \#_i(HV(i) \cup RV(i,j), j)$ times. We have to compare `Goal2` with every clause in the program. Say the number of free clauses in the program is $|R|$. Then we have to execute $\sum_{i=1}^{k} \sum_{j=0}^{l(i)-1} \#_i(HV(i) \cup RV(i,j), j) \times |R|$ unifications. Under the assumption that the length of the atoms in the program is bounded by a constant, the total time needed in the prediction steps amounts to $\sum_{i=1}^{k} \sum_{j=0}^{l(i)-1} \#\#_i(HV(i) \cup RV(i,j), j) \times |R|$.

Remains to be counted the number of steps in `add_items_df`. Suppose we try to add new items immediately after every completion and prediction step. We organize our agenda and table in such a way that the time needed to look up and insert in the agenda and in the table is linear in the size of the item we want to insert. The length of the term that results after unification of two terms is at most the sum of the length of those two terms. After every completion step, which costs time proportional to the sum of the length of two items, we have to do a lookup and an insert which are also proportional in the sum of the length. Thus, we only have to multiply the number of completion steps by two. The same holds for the prediction steps. Efficient organisation of the agenda and the table implies that the movement from the agenda to the table gets a liitle more complicated. The time complexity of moving all items from the agenda to the table equals precisely the space complexity that we saw in the previous section. The total time needed for all operations together is:

$$\sum_{i=1}^{k} \sum_{j=0}^{l(i)-1} \#\#_i(HV(i) \cup RV(i,j), j) \times \sum_{i=1}^{k} \#_i(HV(i), l(i)) \; +$$

$$\sum_{i=1}^{k} \sum_{j=0}^{l(i)-1} \#_i(HV(i) \cup RV(i,j), j) \times \sum_{i=1}^{k} \#\#_i(HV(i), l(i)) \; +$$

$$\sum_{i=1}^{k} \sum_{j=0}^{l(i)-1} \#\#_i(HV(i) \cup RV(i,j), j) \times |R| \; +$$

$$\sum_{i=1}^{k}\sum_{j=0}^{l(i)} \#\#_i(HV(i) \cup RV(i,j), j)$$

This can be simplified to:

$$(\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1} \#\#_i(HV(i) \cup RV(i,j), j) \times (\sum_{i=1}^{k} \#_i(HV(i), l(i)) + |R|)) \; +$$

$$(\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1} \#_i(HV(i) \cup RV(i,j), j) \times \sum_{i=1}^{k} \#\#_i(HV(i), l(i)))$$

We can use this formula to estimate the time complexity of the sample program PATH in the previous section:
$\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1} \#\#_i(HV(i) \cup RV(i,j), j) =$

$\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1} \#_i(HV(i) \cup RV(i,j), j) = \mathcal{O}(|V|^3)$

$\sum_{i=1}^{k} \#\#_i(HV(i), l(i)) = \sum_{i=1}^{k} \#_i(HV(i), l(i)) = \mathcal{O}(|V|^2 + |E|)$

$|R|$, the number of free clauses in the program, is $|E|$.

The time complexity is $\mathcal{O}(|V|^3 \times (|V|^2 + |E|)) = \mathcal{O}(|V|^5)$, because $|E| \leq |V|^2$.

This bound is a very high one. One would expect that the bound would not be higher than $\mathcal{O}(|V|^2)$. One of the reasons is that the meta-interpreter we have described is not very efficient. Later we will define a more efficient interpreter. Before we do this we first define two classes of programs: the *well-moded* and *nicely moded* programs. The reason to do this is to make the formulas and the proofs easier. Theoretically, the restriction to nicely moded programs is not necessary. Without it, however, things get nasty and complicated. The main point of this chapter is that the methods presented are interesting. The exact details are less important. Therefore, we try to keep things simple.

5.6 Well-moded and Nicely Moded Programs

This section is an adapted version of Apt and Pellegrini (1994). We introduce modes. Modes indicate how the arguments of a relation behave.

5.6.1. DEFINITION. Consider an n-ary relation symbol p. By a *mode* for p we mean a function m_p from 1,...,n to the set $\{+, -, ?\}$. If $m_p(i) = '+'$ we call i an *input position* of p. If $m_p(i) = '-'$ we call i an *output position* of p and if $m_p(i) = '?'$ we call i a *throughput position* of p. By a *moding* we mean a collection of modes, each for a different predicate symbol.

We write m_p as $p(m_p(1), \ldots, m_p(n))$. For example, `append(-,?,+)` denotes a ternary predicate `append` with the first position moded as output, the second position as throughput and the third as input. The definition of moding assumes one mode

per predicate in a program. Multiple modes can be obtained by simply renaming the predicates. We will assume that this has been done and that every predicate has a fixed mode associated with it.

Intuitively, the modes indicate how the arguments of a predicate should be used. The given, known, arguments should be put in the input positions and the variables whose value must be computed are in the ouput positions. The throughput positions have an input part and an output part, the arguments are partially instantiated.

This can be illustrated with an example. Consider the predicate `append(-,?,+)`:

```
append([],T,T).
append([A|R],L,[A|T]) :-
        append(R,L,T).
```

and the query `append(P,[3|Q],[1,2,3,4,5])`. In this query, the third argument is the input. The first argument, `P` is clearly output. The second argument is input and output. The query will only succeed when "`3`" is a member of the input list. So "`3`" is the input part. The value of `Q` is computed and is output (`[4,5]`). Therefore we call the second argument a throughput position.

To simplify the notation, we write all atoms as $p(in, o, t)$, where in is a sequence of terms filling the input positions of p, o is a sequence of terms filling the output positions, and t is a sequence of terms filling the throughput positions.

- A query $p_1(in_1, o_1, t_1), \ldots, p_n(in_n, o_n, t_n)$ is called *well-moded* if for $j \in [1, n]$

$$Var(in_j) \subseteq \bigcup_{k=1}^{j-1} Var(o_k) \cup Var(t_k)$$

- A clause

$$p_0(in_0, o_0, t_0) \leftarrow p_1(in_1, o_1, t_1), \ldots, p_n(in_n, o_n, t_n)$$

  is called *well-moded* if for $j \in [1, n]$

$$Var(in_j) \subseteq (Var(in_0) \cup \bigcup_{k=1}^{j-1} (Var(o_k) \cup Var(t_k))) \wedge$$

$$(Var(o_0) \cup Var(t_0)) \subseteq (Var(in_0) \cup \bigcup_{k=1}^{n} (Var(o_k) \cup Var(t_k)))$$

- A program is called well-moded if every clause of it is.

Thus, a query is well-moded if

- in_1 is ground, and every variable occurring in an input position of an atom ($j \in [2, n]$) occurs in an output or a throughput position of an earlier ($k \in [1, j-1]$) atom.

And a clause is well-moded if

- Every variable occurring in an input position of a body atom ($j \in [1,n]$) occurs either in an input position of the head (in_0) or in an output or a throughput position of an earlier ($k \in [1, j-1]$) body atom,
- Every variable occurring in an output or a throughput position of the head occurs either in an input position of the head, or in an output or a throughput position of a body atom.

Observe that no difference is made between output and throughput positions as yet. This difference is important in the definition of *nice modedness*.

- A query
$$p_1(in_1, o_1, t_1), \ldots, p_n(in_n, o_n, t_n)$$
is called *nicely moded* if it is well-moded *and* for all $j \in [1,n]$, o_j is a sequence of variables, i.e. all output positions are filled with variables *and*
$$Var(o_j) \cap (\bigcup_{k=1}^{j-1}(Var(in_k) \cup Var(o_k) \cup Var(t_k))) = \emptyset$$
- A clause
$$p_0(in_0, o_0, t_0) \leftarrow p_1(in_1, o_1, t_1), \ldots, p_n(in_n, o_n, t_n)$$
is called *nicely moded* if it is well-moded *and* for all $j \in [1,n]$, o_j is a sequence of variables, i.e., all output positions are filled with variables *and*
$$Var(o_j) \cap (\bigcup_{k=0}^{j-1}(Var(in_k) \cup Var(t_k)) \cup \bigcup_{k=1}^{j-1} Var(o_k)) = \emptyset$$
- A program is called nicely moded if every clause of it is.

Thus, a query is nicely moded if

- it is well-moded and the output positions are occupied by single "fresh" variables.

And a clause is nicely moded if

- it is well-moded and in all body atoms, the output positions are occupied by single variables that are either "fresh" or occur in the output position of the head.

We go back to the predicate `append(-,?,+)`. We have two clauses:

```
append([],T,T).
append([A|R],L,[A|T]) :-
        append(R,L,T).
```

and suppose the query is `append(P,[3|Q],[1,2,3,4,5])`. The program and the query are nicely moded. Observe that they are also nicely moded under the moding `append(?,?,+)`. Saying that a position is output gives more information than saying it is throughput. If we mode our programs, our aim is to have as many output positions as possible.

The definition of nice-modedness differs from the one in Apt and Pellegrini (1994) in a few respects. In their definition there are no throughput variables, and the

restriction that the arguments are filled with single variables is omitted. In Apt and Pellegrini (1994) the input and output variables a_i and in_i must be disjoint and nice-modedness is not a stronger version of well-modedness, it is just a separate notion. In our definition, every nicely-moded program is also well-moded. Because we want to define the same concept, (fresh variables) as Apt and Pellegrini (1994), we use the same term (nice-modedness) although this could lead to confusion.

We can prove the following results if we have nicely moded queries and programs:

- All computed answer substitutions are ground (Apt and Pellegrini 1994)
- All subgoals in the proof procedure have ground input arguments (follows from the previous result and from the definitions).
- All subgoals in the proof procedure have uninstantiated output arguments (follows from the definitions).

5.7 A Lower Estimate

The upper bound on the time complexity we gave on page 79 was very high, merely because we used a very simple interpreter. It is possible to write an interpreter that is much more efficient. This will be done in the next chapter. One of the main improvements is that we eliminate the unification in the completer step. This can be done as follows. Every item with a non-empty body is associated with two keys. The first key tells how the clause was "called". When we have proved the rest of the body, this key can be used to find all items that are waiting for this solution. The second key tells what items can be a solution for the first atom in the body. The keys are computed in the prediction step. With this usage of keys we can immediately see whether a given solution "fits" a waiting item. A disadvantage is that the number of items grows a little bit: we not only store the substitutions for the head variables after some part of the body has been proven, but we also store the substitutions when nothing has been proved yet (the "call").

We assume that our programs and queries are nicely moded. Suppose clauses are of the form C_i :- $B_{i1}, \ldots B_{in}$. Instead of HV and RV we now define the following sets of variables:

5.7.1. DEFINITION. $HVin(i)$ is the set of variables occurring in an input position of C_i.
$HVout(i)$ is the set of variables occurring in an output position of C_i.
$HVthrough(i)$ is the set of variables occurring in a throughput position of C_i.

$$W(i,j) = ((Var(B_{i1}) \cup \ldots \cup Var(B_{ij})) \setminus HVin(i)) \cap \\ (Var(B_{i(j+1)}) \cup \ldots \cup Var(B_{in}) \cup HVout(i) \cup HVthrough(i))$$

W is the set of variables that have been instantiated. It consists of the relevant body variables and the instantiated out- and throughput variables in the head (they occur in the first part of the body). The input variables in the head are excluded.

$$V(i,j) = Var(B_{i(j+1)}) \setminus (Var(B_{i1}) \cup \ldots \cup Var(B_{ij}) \cup HVin(i))$$

V is the set of uninstantiated variables that will be instantiated in the next step.

We introduced the functions $\#_i(S,j)$ and $\#\#_i(S,j)$. There we counted the number of different $\sigma_n \circ \ldots \circ \sigma_1 \circ f$. Two substitutions were regarded as different when they differed in their substitutions for the variables in S. Now we are going to extend the definition of $\#\#_i(S,j)$, we are going to allow lists of pairs of variable sets and indices. We define $\#_i([\langle X_1, j_1\rangle, \ldots, \langle X_n, j_n\rangle])$. First we determine the highest index in $j_1 \ldots j_n$. Let's say the highest index is k. Then we are going to count the number of different $\sigma_k \circ \ldots \circ \sigma_1 \circ f$. But instead of applying $\sigma_k \circ \ldots \circ \sigma_1 \circ f$ to the variables in S, we now apply $\sigma_{j_1} \circ \ldots \circ \sigma_1 \circ f$ on the variables in X_1, $\sigma_{j_2} \circ \ldots \circ \sigma_1 \circ f$ on the variables in X_2 etcetera. We count the number of possible outcomes for all these substitutions together.

The following formula is an upperbound for the time complexity of the program (this is shown on page 100):

$$\sum_{i=1}^{k} \sum_{j=0}^{l(i)-1} nc(i,j) \times \#\#_i([\langle HVin(i) \cup HVthrough(i), 0\rangle, \langle W(i,j), j\rangle, \langle V(i,j), j+1\rangle])$$

We define a simplified notation here where we drop the j from the function. For a variable X, $\#_i(X) = \#_i(X,0)$ if $X \in HVin(i)$. If X is not in $HVin(i)$, then $\#_i(X) = \#_i(X,j)$ with $X \in Var(B_{ij})$ but not $X \in Var(B_{i1}) \cup \ldots \cup Var(B_{i(j-1)})$ (j is the position of the first occurrence of X). It is always the case that $\#_i(X,j) \leq \#_i(X)$.

In the short notation the complexity formula is:

$$\sum_{i=1}^{k} \sum_{j=0}^{l(i)-1} nc(i,j) \times \#\#_i([\langle HVthrough(i), 0\rangle, \langle HVin(i) \cup W(i,j) \cup V(i,j)\rangle])$$

where $nc(i,j)$ is the number of clauses whose head has the same predicate name as B_{ij} has. When all free predicates are ground facts and have no throughput variables, we can leave out $nc(i,j)$. This is discussed on page 100. The correctness of the formula will be proved in the next chapter.

5.8 A Small Recapitulation

We repeat here all assumptions and results. We give a method to estimate the time complexity of PROLOG THEOREM PROVING FOR CORE PROGRAM P (introduced on page 73). We make the following assumptions:

- we consider pure Prolog, i.e., there are no extra-logical predicates.
- the program is nicely moded (see section 5.6).

The time complexity of PROLOG THEOREM PROVING FOR CORE PROGRAM P is

$$\sum_{i=1}^{k} \sum_{j=0}^{l(i)-1} nc(i,j) \times \#\#_i([\langle HVthrough(i), 0\rangle, \langle HVin(i) \cup W(i,j) \cup V(i,j)\rangle])$$

where $nc(i,j)$ is the number of clauses whose head has the same predicate name as B_{ij} has.
The following remarks can be made:

- the sets $HVthrough(i)$, $HVin(i)$, $W(i,j)$ and $V(i,j)$ are defined on page 82
- we count the number of substitutions for variables in the items generated by the meta-interpreters from page 69
- all substitutions are "alphabetically different"
- if the sum is infinite, the search space is infinite. We can not predict whether the program will terminate or not in this case.
- the length of the clauses in the program is bounded by a constant.
- if B_{ij} is a fixed predicate $nc(i,j)$ is a constant. If B_{ij} is free, we have to count the number of clauses for the predicate.
- when the clauses of B_{ij} are ground facts and have no throughput variables, we can leave out $nc(i,j)$ from the formula.

5.9 Two Examples

We repeat the sample program PATH here with a moding that makes it nicely moded.

```
% Sample program PATH

mode(path(+,-)).
path(X,Z) ::-
       [path(X,Y),
        edge(Y,Z)].
path(W,W) ::- [].

mode(edge(+,-)).
edge(a,b) ::- [].
edge(b,c) ::- [].
edge(c,a) ::- [].
edge(c,d) ::- [].

?- path(a,X).
```

There are no throughput positions, therefore we estimate

$$\sum_{i=1}^{k} \sum_{j=0}^{l(i)-1} nc(i,j) \times \#\#_i([\langle HVin(i) \cup W(i,j) \cup V(i,j)\rangle])$$

Application to the program gives
$\#\#_0(\{X,Y\}) + nc(0,1) \times \#\#_0(\{X,Y,Z\})$
because
$HVin(0) = \{X\},\ W(0,0) = \emptyset,\ V(0,0) = \{Y\},\ W(0,1) = \{Y\},\ V(0,1) = \{Z\}$

Observe that the complexity formula is not conscious of facts, it only considers rules. The only place where facts are counted is in $nc(i,j)$.

$|V|$ and $|E|$ are the number of vertices and edges (the length of these is constant). $\#_0(\{X\}) = 1$ because X always equals the first argument in the query.

$\#_0(\{Y\}) = \#_0(\{Z\}) = |V|$. The time complexity is

$$\mathcal{O}(|V|) + \mathcal{O}(|E| \times |V|^2) = \mathcal{O}(|E| \times |V|^2)$$

Because there must always be an edge between `Y` and `Z`, $\#_0(\{Y,Z\}) = |E|$. We can reduce the complexity to $\mathcal{O}(|V| + |E|^2)$.

Because the clauses for `edge` are ground and the predicate has no throughput variables we can leave out $nc(i,j)$. This gives a time complexity of $\mathcal{O}(|V| + |E|)$. Because $|E| > |V|$ (every vertex `Y` must have an incoming edge), we can further reduce to $\mathcal{O}(|E|)$, which is a lot better than the complexity we obtained with the inefficient meta-interpreter ($\mathcal{O}(|V|^5)$).

A second example is the following. Suppose we have the Prolog program in Figure 5.3 (two-column style) and a query

```
?- s([a,woman,saw,a,man,with,a,telescope],X).
```

For the moment, we want to express the complexity in the length of the input string only and consider the grammar and the lexicon as fixed. Therefore, $nc(i,j)$ is $\mathcal{O}(1)$ for all i and j.

```
s(A,C)    :-                  pp(A,C)   :-
        np(A,B),                      prep(A,B),
        vp(B,C).                      np(B,C).
s(A,E)    :-                  v(A,B)    :-
        det(A,B),                     lex(v,A,B).
        n(B,C),               det(A,B)  :-
        v(C,D),                       lex(det,A,B).
        np(D,E).              n(A,B)    :-
vp(A,C)   :-                          lex(n,A,B).
        v(A,B),               prep(A,B) :-
        np(B,C).                      lex(prep,A,B).
np(A,C)   :-                  lex(v,[saw|A],A).
        np(A,B),              lex(det,[a|A],A).
        pp(B,C).              lex(n,[woman|A],A).
np(A,C)   :-                  lex(n,[man|A],A).
        det(A,B),             lex(n,[telescope|A],A).
        n(B,C).               lex(prep,[with|A],A).

mode(pred/2, [+,-]).  for all pred/2
mode(lex/3, [+,+,-]).
```

Figure 5.3: DCG I

We consider the clause with the longest body, the complexity of other clauses is less than the complexity of this clause.

```
s(A,E) :-
        det(A,B),
        n(B,C),
        v(C,D),
        np(D,E).
```

The complexity is:
$\#\#(\{A,B\}) + \#\#(\{A,B,C\}) + \#\#(\{A,C,D\}) + \#\#(\{A,D,E\})$

This equals: $\mathcal{O}(n^3) + \mathcal{O}(n^4) + \mathcal{O}(n^4) + \mathcal{O}(n^4)$.

For the clause:

```
det(A,B) :-
        lex(det,A,B).
```

The complexity is
$\#\#(\{A,B\}) = \mathcal{O}(n^3)$

The total complexity is $\mathcal{O}(n^4)$.

A "trick" to get a more efficient algorithm is a replacement of the basic datastructure, the list, by indices that represent positions in the list. This is illustrated in the Prolog program in Figure 5.4. Suppose our query is `?- s(0,8)`..

```
s(A,C)    :-              lex(det,A,B) :-
        np(A,B),                  input(a,A,B)
        vp(B,C).          lex(n,A,B)   :-
vp(A,C)   :-                      input(man,A,B).
        v(A,B),           lex(n,A,B)   :-
        np(B,C).                  input(woman,A,B).
np(A,C)   :-              lex(n,A,B)   :-
        np(A,B),                  input(telescope,A,B).
        pp(B,C).          lex(v,A,B)   :-
np(A,C)   :-                      input(saw,A,B).
        det(A,B),         lex(prep,A,B)   :-
        n(B,C).                   input(with,A,B).
pp(A,C)   :-              input(a,0,1).
        prep(A,B),        input(woman,1,2).
        np(B,C).          input(saw,2,3).
v(A,B)    :-              input(a,3,4).
        lex(v,A,B).       input(man,4,5).
det(A,B)  :-              input(with,5,6).
        lex(det,A,B).     input(a,6,7).
n(A,B)    :-              input(telescope,7,8).
        lex(n,A,B).
prep(A,B) :-
        lex(prep,A,B).
```

Figure 5.4: DCG II

If we assume that it takes constant time to compare two numbers, then the complexity of this example is not $\mathcal{O}(n^4)$ but $\mathcal{O}(n^3)$. The number of possible substitutions is the same, but the length has decreased because we do not use lists but numbers. If we assume that the length of the substitutions is a constant we get $\mathcal{O}(n^3)$ for the complexity of the program. Chapter 7 contains two big examples. One finds there

implementations of recognizers for the fragments of the Lambek calculus described in section 1.2.

5.10 Meta-logical Predicates

Everything that has been said up till now applies to pure Prolog only. But what if we want to add meta-logical predicates like negation, cut, assert and retract? These things are needed in practice.

Adding **negation** to the language is not trivial when one uses Earley deduction. The Earley interpreter deals with positive information only. If there is a goal that has to be proven we generate active items which will, if possible, result in solutions. It is not possible, however, to decide that there will be no more solutions for a given goal. But this is precisely what we need for negation. We want to know whether we can expect a solution in the future or not. When we are sure that there will be no solution for the negated goal, we can move the dot over the negated goal. There are two solutions for allowing negation in tabled deduction. The first is that we allow only stratified programs. A stratified program has only recursions that are "negation-free". In stratified programs it is impossible that we have to prove `G` in order to prove `not G`.

The standard example of a program that is not stratified is in Figure 5.5.

```
win(X) :-
        move(X,Y),
        not(win(Y)).

move(a,a).
move(a,b).
move(b,c).
```

Figure 5.5: Example program `win`

This is a two-player game. A player wins if the other player can not move. Position b is a winning position: you can move to c and your enemy is stuck. But what about position a? This is neither a winning nor a losing position because both players will stay in a forever.

This program looks reasonable at first sight. The strange thing about it, however, is that if we fill in `move(a,a)` in the win rule we get "not win(a) implies win(a)". From a logical point of view this is a very strange rule. Therefore it is not a bad idea to forbid non-stratified programs.

When we forbid non-stratified programs, we can create a new agenda when a negated goal is called. We add and delete items from this new agenda until it is empty (we use the same table in the meanwhile). When the agenda is empty we look in the table whether the goal has been proved. If it has not been proved, the negation of the goal is true and we carry on with the old agenda.

If we want to allow non-stratified programs we can introduce a third truth value, the value *undefined*. This approach is followed in (Warren 1991) and is not discussed further here.

The second meta-logical predicate we discuss here is the **cut (!)**. The cut is a means to direct the proof search in standard interpreters with backtracking. The effect of a cut is two-fold:

- the interpreter does not backtrack in the part of the body before the cut and
- the interpreter does not try other rules in the program.

In the Earley interpreter only the first effect makes sense. In order to avoid confusion it is better to replace the cut by a predicate called `solve_once`. Suppose we have a list and we want to remove all occurences of two arbitrary members. Instead of the program in Figure 5.6

```
delete2(Inlist,Outlist) :-
        member(A,Inlist),!,
        delete(A,Inlist,List),
        member(B,List),!,
        delete(B,List,Outlist).
```

Figure 5.6: Example program `delete` I

we could write the program in Figure 5.7.

```
delete2(Inlist,Outlist) :-
        solve_once(member(A,Inlist)),
        delete(A,Inlist,List),
        solve_once(member(B,List)),
        delete(B,List,Outlist).
```

Figure 5.7: Example program `delete` II

The predicate `solve_once` can be implemented by (arbitrarily) throwing away all active items but one for some position of the dot. The second effect, not trying other rules, does not make sense in Earley Deduction. This will be clear from the following example. In standard Prolog one can write the program `not` as in Figure 5.8.

```
not(X) :-
        call(X),!,fail.
not(X).
```

Figure 5.8: Example program `not`

When this program is executed by the Earley interpreter, it will simply succeed for every X because of the fact `not(X)`.

The meta-logical predicates **assert** and **retract** are dangerous. They can change the truth value of a goal. The storage of all proof attempts presupposes that the truth values can not change. It does not seem very attractive to add these two meta-logical predicates. On the other hand we can use assert and retract to get a more efficient implementation. If we want to implement an efficient DCG we can write (Figure 5.9):

```
recognize(X) :-
        retractall(input(_,_,_)),
        assertall(X,0,End),
        s(0,End).
assertall([],End,End).
assertall([H|T],I,End) :-
        J is I + 1,
        assert(input(H,I,J)),
        assertall(T,J,End).
```

Figure 5.9: Example program `assert`

This is a trick that makes the program in Figure 5.4 suited for arbitrary input.

5.11 Further Research

As said in the introduction, the approach taken in this chapter is modest. We have proven an upper bound for the time complexity of Prolog programs in a pretty straightforward way. The bad thing is that the bound is often higher than one would desire. Pereira (1993, p. 547) states:

> "it was felt that the cost of a procedure call in a reasonable programming language should not depend on the sizes of the actual parameters"

In the previous we saw that the complexity *does* depend on the size of the parameters. There are two sources for this. First, we perform the occur check (the quote from Pereira is from an argument against the occur check). Secondly, we make copies of the parameters when we store them in the tables and we have to compare parameters all the time. This comparison is often not necessary. E.g. if we have a DCG as mentioned in Figure 5.3 there will be many copies of tails of the input sentence. Instead of copying this list we would like to copy pointers to positions in this list. If we want to compare two terms in this case, we only have to see whether two pointers point to the same address. And we do not have to compare the two lists. In a practical system we could do the following. We first look whether two pointers point to the same address. If they do, then we are sure they are identical and we don't have to compare or copy anything. If they point to different addresses we have to compare them. It is still possible that they are the same.

```
firstpart([a,b]).
firstpart([a]).
secondpart([c]).
secondpart([b,c]).
pred1 :-
        firstpart(X),
        secondpart(Y),
        append(X,Y,Z),
        pred2(Z).
```

We have to compute twice whether `pred2([a,b,c])`, but the two terms `[a,b,c]` will be represented internally in a different way. Theoretically the "trick" does not help us much, but in practice it can be an improvement. In practice the size of the parameters will often be eliminated.

5.12 Existing Implementations of Earley Interpreters

Currently there are (at least) three implementations of the Earley interpreter. The first is a very experimental implementation by the author. This implementation can be found at `"http://www.fwi.uva.nl/~aarts/prolog.html"`. The interpreter has been written in Prolog and runs under Quintus, Sicstus and SWI Prolog. A nice feature is high-level tracing, where we can inspect proof trees after the execution of the program. The high-level tracer stimulates a declarative programming style. The second implementation is SLG. This is also a meta-interpreter in Prolog. It supports the third truth value for non-stratified programs. It has been developed by W. Chen and D.S. Warren. The third implementation is XSB. XSB has been developed at the University of New York at Stony Brook by D.S. Warren et al. This interpreter allows both Earley and standard deduction. The idea is that one can use the standard interpreter for simple, deterministic, predicates where tabling is a useless overhead. Complex non-deterministic predicates can be tabled. XSB has been written in C and is a fast Prolog interpreter. The home page of the XSB Group (where XSB and SLG can be found) is: `"http://www.cs.sunysb.edu/~sbprolog/"`.

Chapter 6

Proof of the Time Complexity Result

6.1 A More Efficient Earley Interpreter

In this chapter we describe in detail a more efficient Earley interpreter. Furthermore we prove an estimate for the time complexity of this interpreter. This estimate has been presented without proof in the previous chapter.

The datastructure used in the one-solution and all-solutions interpreters in the previous chapter, was the *item*, that consisted of a clause with a remainder of the body. We will use two integers that represent the clause number and the current position in the body. We assume, in analogy with the Earley algorithm, that a *dot* indicates our progress in the body. Furthermore, we add a substitution for two kinds of variables:

- variables that occur in the head, the head variables, and
- variables that occur in the body before *and* after the dot, but not in the head: the *relevant* body variables.

Variables that occur only before the dot are irrelevant because they have no influence on the proof of the rest of the body. The variables occurring after the dot only are simply not instantiated yet, there is no substitution. These are precisely the variable sets HV and RV defined on page 76. The substitutions for the head variables and the relevant body variables are stored in two lists.

We assume that we have a preprocessor that builds two program tables called "head" and "body" in advance. Table entries in the "head" table contain

- a clause number (integer)
- the length of the body (integer)
- the head of the clause (atom)
- the head variables (list of variable names)

Table entries in the "body" table contain

- a clause number (integer)
- the position of the dot in the body (integer)
- the length of the body (integer)

- the head variables (list of variable names)
- the member of the body right after the dot, say `Bi` (an atom)
- the relevant body variables *before* `Bi`, i.e. a list of variables occurring in `B1,...,B(i-1)` *and* in `Bi,...,Bn` in some fixed order (list of variable names)
- the relevant body variables *after* `Bi`, i.e. a list of variables occurring in `B1,...,Bi` *and* in `B(i+1),...,Bn` in some fixed order (list of variable names)

In order to keep things simple we use two "tricks" in the program table. The first is that we add a rule for the query. Suppose our query is `Q1,...,Qn`. Then we add to the program the rule `success :- Q1,...,Qn`. Our new query is now `success`. This trick reduces the number of initialization steps in the interpreter. The second trick is that facts are represented as rules with the body `true`. This also simplifies the interpreter.

In the interpreter we create three kinds of objects: *active items*, *inactive items* and *extended active items*. These objects are stored in three tables. There is also a table containing the goals that are "called" together with a key for faster lookup in other tables. The contents of the tables are described precisely below.

Active items replace the objects called items in the variable-free Earley interpreter. Active items are six-tuples of

- a clause number (integer)
- the position of the dot in the body (integer)
- the length of the body (integer)
- a key representing the head as it was instantiated before any members of the body were proved, as represented by $f(C)$ in Figure 5.1 (integer)
- a list of substitutions for the head variables (list of terms)
- a list of substitutions for the relevant (body) variables (list of terms)

Inactive items are active items whose dot is at the end of the body, i.e. they are finished. Inactive items are triples. They look like active items but without the body position, body length and the substitution for the relevant variables:

- a clause number (integer)
- a key like in the 4th slot of the active items (integer)
- an instantiated head of the clause, the solution (an atom)

Extended active items are active items with one more slot. From an active item we can compute the next member of the body that has to be proved. This goal has to match some clause in the program. Through unification with the head of this clause, the goal is instantiated further. After unification, the goal is looked up in the call table. If the table contains the goal (or an alphabetic variant) we read the key from the table. If the table does not contain the goal, we add it to the table and generate a new key. Now we have a key and we put it in the extra slot in the extended active item.

Therefore, extended active items consist of

- key representing the body member to be proven (integer)
- clause number (integer)

- the position of the dot in the body (integer)
- the length of the body (integer)
- a key representing the head as it was instantiated before any members of the body were proved (integer)
- a list of substitutions for the head variables (list of terms)
- a list of substitutions for the relevant variables (list of terms)

The call table contains:

- clause number (integer)
- a substitutions for the head variables (list of terms)
- a key (integer)

Besides the tables just mentioned, we have an *agenda* of unprocessed active items.

The algorithm is described in two ways. In the text that follows we give an implementation in an imperative Pascal-like language. We will present some pseudo code and will do a complexity analysis based on this pseudo code. In Appendix A one finds Prolog code that does exactly the same. Readers familiar with Prolog can look there. The code in the appendix is no pseudo code but real code.

In our imperative language we define a program called *main*. This program initializes the agenda and then takes active items from this agenda repeatedly. For each active item one of the three following procedures is executed.

- When the body position equals the body length (lines 14-29 in the main program), the active item is in fact an **inactive item**. We remove the body length, body position and the (empty) list of relevant variables. From the instantiations for the head variables and the table entry in the head table we compute the instantiated head, the solution. Then we put the solution together with the key and the clause number in the table of inactive items. The completer is started. The completer searches in the table of extended active items for an item whose key in the first slot equals the key of the inactive item. If the keys are equal, then the inactive item is a solution for the extended active item. The dot is moved and a new substitution for the relevant variables is looked up in the body table. The completer generates new active items which are put on the agenda of unprocessed active items. (This procedure is the first clause of `predict_or_complete` in the appendix).
- Prolog **facts** are represented as clauses with the body `true`. When the active item's clause number points to such a clause, we increment the body position with one and put the active item back on the agenda (lines 31-33 in program *main*). One step later the procedure described in the previous item will take care of the active item. (This procedure is the second clause of `predict_or_complete` in the appendix).
- In other cases (the procedure predict) we look twice in the program table. First we look what goal has to be proved next. Then we try to unify that goal with the head of some clause. When that is possible we look in the call table whether the goal has been "called" earlier. There are two cases:

- the goal has been called earlier. We read the key from the table, add the key to the active item and get an extended active item. We try to add the item to the table of extended active items. If that is possible, then it has not been processed earlier. The completer will try to combine the extended active item with inactive items, leading to new active items.
- the goal has not been called earlier. What follows can be seen as a predictor step. We create a new entry in the call table and generate a new key. We add to the extended active items table a new entry, the active item extended with the key. Furthermore we add an active item with body position 0 to the agenda.

One of these two things will be done for all clauses whose head unifies with the given goal. This procedure is the third clause of `predict_or_complete` in the appendix.

When the agenda is empty we are finished.

We present here some pseudo code that does what has just been described. We assume we have the following primitives.

push(Item,Agenda). Push an item on the agenda and put it in the table of active items.

push_init(Item,Agenda). If the item is not in the table of active items, add it and push it on the agenda. (init = If Not In Table).

pop(Item,Agenda). Pop an item from the agenda.

For operations on tables we have three routines.

insert(table_name(Var1,Var2,....,Varn)). Insert something in a table.

for all <Vari,Var(i+1),....,Varn⟩ **such that** table_name(Var1,Var2,....,Varn) **begin end**. For all table entries for which some condition holds do something.

get <Vari,Var(i+1),....,Varn⟩ **from** table_name(Var1,Var2,....,Varn). Search for the table entry table_name(Var1,Var2,....,Varn), the first variables are instantiated, the latter are not and should be returned.

There are three kinds of boolean expressions referring to the content of the tables:

in_table(table_name(Var1,Var2,....,Varn)) is true if there exists a table item that is an alphabetic variant.

there is <Vari,Var(i+1),....,Varn⟩ **such that** table_name(Var1,Var2,....,Varn). The solution must be unique. This is the "deterministic" variant of **for all . . . such that**.

If we don't need access to the variables we have a shorter notation we replace the variables that "don't care" by an underscore (_).

Elementary procedures and boolean expressions that have nothing to do with the agenda or with the tables are:

assign(A,B). Assigns a value.
unify(A,B).
unifiable(A,B). Both after unify and after unifiable A and B are instantiated to their most general unifier. Unify is a statement, unifiable is a test.
freeze_term(A,B). freeze_term makes a copy of term A, replaces the free variables by special symbols and assigns the frozen term to B. This is used to test whether two terms are alphabetic variants. Two terms are alphabetic variants if they are identical after freezing.

The pseudocode of the interpreter follows here. The procedure *predict* and the program *main* have been defined as follows:

```
    procedure predict(Agenda,Cn,J0,N,Key,HV,RV,LastKey);


    Jp := J0 + 1 ;
    get <HVu,Bi,RVu,RV_new> from body(Cn,Jp,N,HVu,Bi,RVu,RV_new) ;
5   assign(HVu,HV) ;
    assign(RVu,RV) ; % now we know Bi
    for all <Cn_son,M,Head1,HV_son> such that
    head(Cn_son,M,Head1,HV_son) and unifiable(Head1,Bi)
    begin
10      freeze_term(HV_son,HV_son_nv) ;
        if there is <Key_son> such that called(Cn_son,HV_son_nv,Key_son)
        then
            if not in_table(ext_act_item(Key_son,Cn,J0,N,Key,HV,RV))
            then
15          begin
                insert(ext_act_item(Key_son,Cn,J0,N,Key,HV,RV));
                for all <Solution> such that
                inactive_item(Cn_son,Key_son,Solution)
                begin
20                  unify(Solution,Bi) ;
                    push_init(active_item(Cn,Jp,N,Key,HV,RV_new),Agenda)
                end
            end
        else
25          LastKey := Lastkey + 1 ;
            Key_son := Lastkey ;
            insert(called(Cn_son,HV_son_nv,Key_son)) ;
            insert(ext_act_item(Key_son,Cn,J0,N,Key,HV,RV));
            push(active_item(Cn_son,0,M,Key_son,HV_son,[ ]),Agenda)
30  end
```

```
    program main ;


    Agenda := nil ;
    get <N> from body(0,_,N,_,_,_,_) ;
5   insert(called(0,success,0)) ;
    LastKey := 0 ;
    push(active_item(0,0,N,0,[ ],[ ]),Agenda) ;

    while Agenda not nil do
10  begin
        pop(active_item(Cn,J0,N,Key,HV,RV),Agenda) ;
        if J0 = N
        then
        begin
15          get <Head,HVu> from head(Cn,N,Head,HVu)
            assign(HVu,HV), % now we know Head (the solution)
            insert(inactive_item(Cn,Key,Head)) ;
            for all <Cn_mom,J1,M,Key_mom,HV_mom,RV_mom> such that
            ext_act_item(Key,Cn_mom,J1,M,Key_mom,HV_mom,RV_mom)
20          begin
                Jp := J1 + 1 ;
                get <HV_momu,Bi,RV_momu,RV_mom_new> from
                body(Cn_mom,Jp,M,HV_momu,Bi,RV_momu,RV_mom_new);
                assign(HV_momu,HV_mom) ;
25              assign(RV_momu,RV_mom) ; % now we know Bi
                unify(Head,Bi); % changes HV_mom and RV_mom_new
                push_init(active_item(Cn_mom,Jp,M,Key_mom,HV_mom,
                    RV_mom_new),Agenda)
            end
        end
30      else {J0 ≠ N}
            if J0 = 0 and N = 1 and body(Cn,1,1,_,true,[ ],[ ])
            then
                push_init(active_item(Cn,1,1,Key,HV,[ ]),Agenda)
            else {item not finished and not a fact}
35              predict(Agenda,Cn,J0,N,Key,HV,RV,Lastkey)
    end
    if in_table(inactive_item(0, 0, success))
    then write('yes') else write('no')
```

If we replace the guard

while Agenda **not nil do**

by

while Agenda **not nil and not in_table**(inactive_item(0, 0, success)) **do**

the interpreter stops after finding the first solution.

6.2 Complexity of the Earley interpreter

In this section we are going to count the number of steps that the algorithm in the previous section takes.

The complexity of our primitive operations is as follows.

push(Item,Agenda)	$\mathcal{O}(\|Item\|)$
push_init(Item,Agenda)	$\mathcal{O}(\|Item\|)$
pop(Item,Agenda)	$\mathcal{O}(1)$
insert(table_name(Var1,Var2,....,Varn))	$\mathcal{O}(\|Var1\| + \ldots + \|Varn\|)$
get ⟨Vari,Var(i+1),....,Varn⟩ from table_name(Var1,Var2,....,Varn)	$\mathcal{O}(\|Var1\| + \ldots + \|Varn\|)$
in_table(table_name(Var1,Var2,....,Varn))	$\mathcal{O}(\|Var1\| + \ldots + \|Varn\|)$
there is ⟨Vari,Var(i+1),....,Varn⟩ such that table_name(Var1,Var2,....,Varn)	$\mathcal{O}(\|Var1\| + \ldots + \|Varn\|)$
unify(A,B)	$\mathcal{O}(\|A\| + \|B\|)$
freeze_term(A,B)	$\mathcal{O}(\|A\|)$
unifiable(A,B)	$\mathcal{O}(\|A\| + \|B\|)$

The number of steps of the program *main* is counted as follows. Suppose our Prolog program is of the following form:

```
H1 :-   ...
..
Hi :-
        Bi1,
        Bi2,
        ...
        Bin.
..
Hk :-   ...
```

$\#_i$ and $\#\#_i$ are defined as usual. We also define the average length $|_i(X,i)|$. $\#\#_i(X,i) = \#_i(X,i) \times |_i(X,i)|$.

Remember that the head variables are $HVin(i) \cup HVout(i) \cup HVthrough(i)$. Relevant body variables are $((Var(B_{i1}) \cup \ldots \cup Var(B_{ij})) \cap (Var(B_{i(j+1)}) \cup \ldots \cup Var(B_{in}))) \setminus (HVin(i) \cup HVout(i) \cup HVthrough(i))$.

We **assume** that the length of the clauses in the body of the Prolog rules is bounded by a constant.

We will only consider lines 9-36 in program *main*. The other lines have a lower complexity. The number of active items is

$$\sum_{i=1}^{k}\sum_{j=0}^{l(i)} \#_i([\langle HVthrough(i) \cup HVin(i), 0\rangle, \langle W(i,j)\rangle])$$

$\#_i([\langle HVthrough(i) \cup HVin(i), 0\rangle])$ is the number of possible `Key`'s,
$\#_i([\langle((Var(B_{i1}) \cup \ldots \cup Var(B_{ij})) \setminus HVin(i)) \cap (HVout(i) \cup HVthrough(i)), j\rangle])$ is the number of possible substitutions for `HV` for that key,
$\#_i([\langle((Var(B_{i1}) \cup \ldots \cup Var(B_{ij})) \setminus HVin(i)) \cap (Var(B_{i(j+1)}) \cup \ldots \cup Var(B_{in})), j\rangle])$ is the number of possible substitutions for `RV`. The *while* loop in lines 10-36 is executed that many times.

The amount of work that has to be done in the *while* loop can be counted as follows. We first look at the procedure *predict*. We will show later that the other two alternatives are cheaper.

So we look now at the amount of time spent for a given active item in the procedure *predict*.

- 4 **get** ⟨HVu,Bi,RVu,RV_new⟩ **from** body(Cn,Jp,N,HVu,Bi,RVu,RV_new) ;
  This costs constant time (under the assumption that the length of Bi is constant).

- 7 **for all** ⟨Cn_son,M,Head1,HV_son⟩ **such that**
  head(Cn_son,M,Head1,HV_son) **and** unifiable(Head1,Bi)

  Here the next thing to be proved is matched with all clauses whose head has the same predicate name. With $nc(i,j)$ (Number of Clauses) we will denote the number of clauses with this predicate name, i.e. the number of clauses whose head has the same predicate name as $Bi(j+1)$. In the worst case, all clauses unify, and the rest of procedure *predict* will be executed $nc(i,j)$ times. The unification costs the length of `Bi`: $|_i(Var(B_{i(j+1)}), j)|$ because we assume that the length of the terms in the head of program clauses is a constant.

- 10 freeze_term(HV_son,HV_son_nv) ; Cost depends on the length of `HV_son`. This length is less than $|_i(Var(B_{i(j+1)}), j)|$

- 11 **if there is** ⟨Key_son⟩ **such that** called(Cn_son,HV_son_nv,Key_son)
  Cost depends on the length of `HV_son_nv`. The same cost as in the previous step: $|_i(Var(B_{i(j+1)}), j)|$. We will assume that this test succeeds and count the steps in the "then" branch. We will count later the steps in the "else" branch.

- 13 **if not in_table**(ext_act_item(Key_son,Cn,J0,N,Key,HV,RV))
  The variables in $V(i,j)$ can have changed through the unification with the head of the clause numbered `Cn_son`. I.e. they can be larger than $|_i(V(i,j), j)|$. But

under the assumption that program clause heads have constant length this difference is a constant. The cost depends on the length of `HV` and `RV`. This equals $|_i(W(i,j) \cup HVin(i) \cup (HVthrough(i) \setminus (Var(B_{i1}) \cup \ldots \cup Var(B_{ij}))), j)|$

- 16 **insert**(ext_act_item(Key_son,Cn,J0,N,Key,HV,RV)); See the previous item.
  $|_i(W(i,j) \cup HVin(i) \cup (HVthrough(i) \setminus (Var(B_{i1}) \cup \ldots \cup Var(B_{ij}))), j)|$

- 17 **for all** ⟨Solution⟩ **such that inactive_item(Cn_son,Key_son,Solution)**
  The number of Solutions is $\#_i(V(i,j), j+1)$. The rest has to be executed that many times.

- 20 unify(Solution,Bi) ;
  Costs at most $|_i(Var(B_{i(j+1)}), j+1)|$ because the length of both `Solution` and `Bi` is smaller than $|_i(Var(B_{i(j+1)}), j+1)|$.

- 21 **push_init**(active_item(Cn,Jp,N,Key,HV,RV_new),Agenda)
  Costs $|_i(Var(B_{i(j+1)}), j+1)|+$
  $|_i(W(i,j) \cup HVin(i) \cup (HVthrough(i) \setminus (Var(B_{i1}) \cup \ldots \cup Var(B_{ij}))), j)|$

The total amount (for the "then" branch, lines 13-23) is

$$\#_i([\langle HVthrough(i), 0\rangle, \langle HVin(i) \cup W(i,j)\rangle]) \times nc(i,j) \times$$
$$(|_i(Var(B_{i(j+1)}), j)|+|_i(W(i,j) \cup HVin(i) \cup (HVthrough(i) \setminus (Var(B_{i1}) \cup \ldots \cup Var(B_{ij}))), j)|+$$
$$(\#_i(V(i,j), j+1) \times$$
$$(|_i(Var(B_{i(j+1)}), j+1)|+|_i(W(i,j) \cup HVin(i) \cup (HVthrough(i) \setminus (Var(B_{i1}) \cup \ldots \cup Var(B_{ij}))), j)|)))$$

because $\{|_i(Var(B_{i(j+1)}), j)| < |_i(Var(B_{i(j+1)}), j+1)|\}$

$$\approx \#_i([\langle HVthrough(i), 0\rangle, \langle HVin(i) \cup W(i,j)\rangle]) \times nc(i,j) \times$$
$$(\#_i(V(i,j), j+1) \times (|_i(Var(B_{i(j+1)}), j+1)|+|_i(W(i,j) \cup HVin(i) \cup (HVthrough(i) \setminus (Var(B_{i1}) \cup$$
$$\ldots \cup Var(B_{ij}))), j)|))$$

$$\approx \#_i([\langle HVthrough(i), 0\rangle, \langle HVin(i) \cup W(i,j), j\rangle, \langle V(i,j), j+1\rangle]) \times nc(i,j) \times$$
$$(|_i(Var(B_{i(j+1)}), j+1)| +$$
$$|_i(W(i,j) \cup HVin(i) \cup (HVthrough(i) \setminus (Var(B_{i1}) \cup \ldots \cup Var(B_{ij}))), j)|)$$

$$\approx \#_i([\langle HVthrough(i), 0\rangle, \langle HVin(i) \cup W(i,j), j\rangle, \langle V(i,j), j+1\rangle]) \times nc(i,j) \times$$
$$(|_i(V(i,j) \cup (Var(B_{i(j+1)}) \cap (Var(B_{i1}) \cup \ldots \cup Var(B_{ij}) \cup HVin(i))), j+1)| +$$
$$|_i(W(i,j) \cup HVin(i) \cup (HVthrough(i) \setminus (Var(B_{i1}) \cup \ldots \cup Var(B_{ij}))), j)|)$$

$$\approx \#_i([\langle HVthrough(i), 0\rangle, \langle HVin(i) \cup W(i,j), j\rangle, \langle V(i,j), j+1\rangle]) \times nc(i,j) \times$$
$$(|_i(W(i,j) \cup HVin(i), j)| + |_i(V(i,j), j+1)| +$$
$$|_i(Var(B_{i(j+1)}) \cap (Var(B_{i1}) \cup \ldots \cup Var(B_{ij}) \cup HVin(i))), j+1)| +$$
$$|_i(HVthrough(i) \setminus (Var(B_{i1}) \cup \ldots \cup Var(B_{ij}))), j)|)$$

$$\approx \#_i([\langle HVthrough(i),0\rangle,\langle HVin(i)\cup W(i,j),j\rangle,\langle V(i,j),j+1\rangle])\times nc(i,j)\times$$
$$(|_i(W(i,j)\cup HVin(i),j)|+|_i(V(i,j),j+1)|+$$
$$|_i(Var(B_{i(j+1)})\cap(Var(B_{i1})\cup\ldots\cup Var(B_{ij})\cup HVin(i))),j)|+$$
$$|_i(HVthrough(i)\setminus(Var(B_{i1})\cup\ldots\cup Var(B_{ij}))),j)|)$$

$$\{(Var(B_{i(j+1)})\cap(Var(B_{i1})\cup\ldots\cup Var(B_{ij})\cup HVin(i)))\subseteq(HVin(i)\cup W(i,j))\}$$

$$\approx \#_i([\langle HVthrough(i),0\rangle,\langle HVin(i)\cup W(i,j),j\rangle,\langle V(i,j),j+1\rangle])\times nc(i,j)\times$$
$$(|_i(W(i,j)\cup HVin(i),j)|+|_i(V(i,j),j+1)|+$$
$$|_i(HVthrough(i)\setminus(Var(B_{i1})\cup\ldots\cup Var(B_{ij}))),j)|)$$

$$\{|_i(HVthrough(i)\setminus(Var(B_{i1})\cup\ldots\cup Var(B_{ij}))),j)|<|_i(HVthrough(i),0)|\}$$

$$\approx \#_i([\langle HVthrough(i),0\rangle,\langle HVin(i)\cup W(i,j),j\rangle,\langle V(i,j),j+1\rangle])\times nc(i,j)\times$$
$$(|_i(W(i,j)\cup HVin(i),j)|+|_i(V(i,j),j+1)|+|_i(HVthrough(i),0)|)$$

$$\approx \#\#_i([\langle HVthrough(i),0\rangle,\langle HVin(i)\cup W(i,j),j\rangle,\langle V(i,j),j+1\rangle])\times nc(i,j)$$

The time needed in the "else" branch (lines 25-29) is obviously less.
The total complexity in *predict* is

$$\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1} nc(i,j)\times\#\#_i([\langle HVin(i)\cup HVthrough(i),0\rangle,\langle W(i,j),j\rangle,\langle V(i,j),j+1\rangle])$$

We have counted here all possible combinations of an extended active item and a complete item. If the inactive item is found before the active item the completer steps are performed in the procedure *predict*. On the other hand, if the active item is found first the completion is done in lines 14-28 of program *main*. Because we have counted all completion steps we do not have to consider lines 14-28 of program *main* anymore.

Sometimes we can leave out $nc(i,j)$ from the formula. This happens when all free predicates

- are ground facts, and
- have no throughput variables.

In this case no unification takes place in lines 7 and 8 of the program. We can look up all the answers. The time needed for this is linear in the number of solutions $\#_i([\langle V(i,j),j+1\rangle])$. Without further proof we claim here that we can leave out $nc(i,j)$ under these circumstances.

6.3 Improved Earley Deduction

When we look at the examples in the previous section an improvement on our interpreter is obvious. Suppose we have a clause

```
s(A,E) :- det(A,B), n(B,C), v(C,D), np(D,E).
```

with mode `mode(s(+,+))`. A simple observation is that the value of the variable E is unimportant until we have to prove `np(D,E)`. If we have two goals, say `s(0,3)` and `s(0,6)`, a lot of work will be done twice. Therefore we introduce the notion *relevant head variable*. Relevant head variables are variables occurring both in the head *and* before the point we have reached in the body. In the example, only A is relevant (and E is a body clause variable). We will not describe in detail how this can be implemented in the interpreter. The main idea, however, is as follows. In the procedure *predict* we look whether a goal has been proved earlier or not. Remember the program line:

11 **if there is** ⟨Key_son⟩ **such that** called(Cn_son,HV_son_nv,Key_son)

The head variables are stored in the "called" table in some standard order, e.g. the order of appearance in the head of a rule. We are going to change the order. Head variables are now ordered in their order of *occurrence in the body*. Formerly we knew that a call had been made earlier when the substitution for the head variables was identical (under alphabetic variants). Now we loosen this. We try to match as much variables as possible from left to right. When we can match some variables, but not all we have a "similar query". We can reuse active items from a call that is similar with the current goal. With the rule `s(A,E) :- det(A,B), n(B,C), v(C,D), np(D,E).` and the goals `s(0,3)` and `s(0,6)` we perform the usual steps for `s(0,3)`. When we try to prove `s(0,6)` we look in the "called" table and find the entry `called(...,[0,3],...)`. We see that the value of A (0) matches and the value of E (6) does not, and that the goals are similar. Now we can copy the active items generated by the similar call `s(0,3)` between `v(C,D)` and `np(D,E)`. This saves us the completer steps before `v(C,D)`. It can be even the case that a head variable is not relevant at all in the body. E.g. the predicate `apply(A/B,C,A)` on page 104. If we have two calls that only differ in `A` we can just compute the result once and copy the result. In general, completer steps are performed only once for every substitution of the relevant head variables.

The result of this improvement is that we only count the relevant head variables and not all head variables, i.e., $HVin(i)$ can be reduced to $HVin(i) \cap (B_{i1} \cup \ldots \cup B_{ij})$. We do not go into details here about this improvement.

Chapter 7

Fragments of L in Prolog

In this chapter we give algorithms for the fragments of the Lambek calculus described in chapter 1.2. These algorithms are given in Earley Prolog. In order to estimate the time complexity of the algorithms we use the method of chapter 5.

7.1 Non-associative Lambek Grammar

The implementation given here is a very compact and simple one. One of the reasons is that we can represent the rules of the calculus very natural in Prolog (the ==> predicate).

7.1.1 Implementation

An example lexicon and query are:

```
mode(lex/2, [+,-]).
lex(marie,np).
lex(slaat,np\(s/np)).
lex(de,np/n).
lex(vervelende,n/(pp/(n\pp))).
lex(jongen,n).

?- combine([marie,slaat,de,vervelende,jongen],S).
```

The recognizer is implemented as follows:

```
?-op(230,xfx,'/').          /* Right division       */
?-op(230,xfx,'\').          /* Left division        */
?-op(250,xfx,'==>').        /* R3 - R4 Derivability */

mode(combine/2, [+,-]).
combine([D1],S) :-
        lex(D1,S).
combine(List,S) :-
        append(Begin,End,List),
        combine(Begin,D1),
        combine(End,D2),
        apply(D1,D2,S).

mode(apply/3, [+,+,-]).
apply(A/B,C,A) :-
        C ==> B.
apply(C,B\A,A) :-
        C ==> B.

mode(==>/2, [+,+]).
A ==> A.
W ==> X/(Y\Z)  :-
        W ==> Y,
        Z ==> X.
W ==> (Z/Y)\X  :-
        W ==> Y,
        Z ==> X.
W/X ==> Y/Z :-
        W ==> Y,
        Z ==> X.
X\W ==> Z\Y :-
        W ==> Y,
        Z ==> X.

mode(append/3, [-,-,+]).
append([],K,K).
append([H|T],K,[H|L]) :-
        append(T,K,L).
```

Observe that this program does not work in standard Prolog although it is semantically correct. In order to make it work we have to replace the variables `Begin` and `End` by `[B|Begin]` and `[E|End]`.

7.1.2 Complexity Analysis

Let's repeat the time complexity for Prolog programs from page 100.

$$\sum_{i=1}^{k} \sum_{j=0}^{l(i)-1} nc(i,j) \times \#\#_i([\langle HVthrough(i), 0\rangle, \langle HVin(i) \cup W(i,j) \cup V(i,j)\rangle])$$

The program is nicely moded. The length of the clauses is bounded by a constant. We want to express the time complexity in

- the length of the query: n.
- the number of clauses of the free predicate `lex(...)`: $|Lex|$. The length of the lexical items is considered as a constant!

We apply the formula on the program.

```
mode(combine/2, [+,-]).
combine([D1],S) :-
        lex(D1,S).
```

$|Lex| \times \#\#(\{D1, S\})$

The first argument in `combine` is always a sublist of the list in the first argument of the query. Because the sublist has length 1, there are n possibilities, and every possibility needs constant space. The second argument in `combine` is always a subtype of some type in the lexicon. The number of subtypes is $|Lex|$. There are $|Lex|$ substitutions for $D1$ of constant length. We have to multiply with $|Lex|$, so we get $|Lex|^2$. Complexity here is $\mathcal{O}(|Lex|^2 \times n)$.

```
combine(List,S) :-
        append(Begin,End,List),
        combine(Begin,D1),
        combine(End,D2),
        apply(D1,D2,S).
```

$\#\#(\{List, Begin, End\}) + \#\#(\{List, Begin, End, D1\}) +$
$\#\#(\{List, End, D1, D2\}) + \#\#(\{List, D1, D2, S\})$

The number of sublists (substitutions for `List`) is n^2. The list can be split at n points. When $D1$ and $D2$ are known, S is fixed.
$\mathcal{O}(n^4) + \mathcal{O}(|Lex| \times n^4) + \mathcal{O}(|Lex|^2 \times n^4) + \mathcal{O}(|Lex|^2 \times n^3) =$
$\mathcal{O}(|Lex|^2 \times n^4)$

```
mode(apply/3, [+,+,-]).
apply(A/B,C,A) :-
        C==>B.

apply(C,B\A,A) :-
        C==>B.
```

Complexity for both clauses:
$\#\#(\{A, B, C\})$

We can freely choose two subtypes. Complexity is $\mathcal{O}(|Lex|^2)$.

```
mode(==> /2, [+,+]).
W==>X/(Y\Z) :-
        W==>Y,
        Z==>X.

W==>(Z/Y)\X :-
        W==>Y,
        Z==>X.

W/X==>Y/Z :-
        W==>Y,
        Z==>X.

X\W==>Z\Y :-
        W==>Y,
        Z==>X.
```

Complexity for all clauses:
$\#\#(\{W,X,Y,Z\}) +$
$\#\#(\{W,X,Y,Z\})$

We can freely choose two subtypes. Complexity is $\mathcal{O}(|Lex|^2)$.

```
mode(append/3, [-,-,+]).
append([H|T],K,[H|L]) :-
        append(T,K,L).
```

$\#\#(\{H,L,T,K\})$

The arguments are sublists in the query. Complexity is $\mathcal{O}(n^4)$.
The complexity of the whole program is $\mathcal{O}(|Lex|^2 \times n^4)$.

7.2 Second Order Lambek Grammar

7.2.1 Implementation

Let's repeat the definition of unfolding here.

7.2.1. DEFINITION. f: $Tp \to ATp^*$, the unfolding function, is defined as follows:
$f((\mathsf{T}_1,\ldots,\mathsf{T}_m \Rightarrow \mathsf{B} \Leftarrow \mathsf{L}_1,\ldots,\mathsf{L}_n)) = \overleftarrow{T_1},\ \ldots\ \overleftarrow{T_m},\ \overrightarrow{B},\ \overrightarrow{L_1},\ \ldots\ \overrightarrow{L_n} \quad (m,n \geq 0)$. We extend the definition to $f : Tp^* \to ATp^*$ by applying f pointwise and concatenating the result.

The words are looked up in the lexicon and the types are unfolded. The arguments, with left and right arrows can be complex. But the arguments of the arguments are primitive because we are in the second order fragment.

The rules that should be applied are:

$\epsilon \rightarrow (\mathsf{T_1},\ldots,\mathsf{T_m} \Rightarrow \overrightarrow{\mathsf{B}} \Leftarrow \mathsf{L_1},\ldots,\mathsf{L_n}), \overleftarrow{T_1},\ \ldots\ \overleftarrow{T_m},\ \overset{\uparrow}{B},\ \overrightarrow{L_1},\ \ldots\ \overrightarrow{L_n} \quad (m,n \geq 0)$

$\epsilon \rightarrow \overleftarrow{T_1},\ \ldots\ \overleftarrow{T_m},\ \overset{\uparrow}{B},\ \overrightarrow{L_1},\ \ldots\ \overrightarrow{L_n},\ (\mathsf{T_1},\ldots,\mathsf{T_m} \Rightarrow \overleftarrow{\mathsf{B}} \Leftarrow \mathsf{L_1},\ldots,\mathsf{L_n})$

All T_i and L_i are primitive. With these definitions in mind we give the following Prolog implementation.

```
% recognize(StringOfWords)
mode(recognize/1, [+]).
recognize(L) :-
        appendp([[]|L],[[]],K),
        match([(up,s)],K).

% match(Top,Bottom)
% Top = ((←p)* ↑p (→p)*) ∪ (→p)*
% Bottom = [[PathCircleToBox],PathBoxToBox,[PathBoxToCircle]]
% ((←p)* ↑p (→p)*) ∪ (→p)* ,List of Words,((←p)* ↑p (→p)*) ∪ (←p)*

mode(match/2, [+,+]).
match([],L) :-
        epsilon(L).
match([H|T1],[[H|T2]|T3]) :-
        match(T1,[T2|T3]).
match([H|T1],[[],[H|T2]]) :-  % PathBoxToBox = []
        match(T1,[[],T2]).
match([H|T1],L) :-
        break(L,M,H,N,K),     % break(Bottom,Bottom1,Bottom2)
                              % splits PathBoxToBox
        epsilon(M),
        match(T1,[N|K]).

% epsilon([[PathCircleToBox],PathBoxToBox,[PathBoxToCircle]])

Succeeds only if
% ((→p)* ,List of Words,((←p)* ↑p (→p)*) ∪ (←p)* or
% ((←p)* ↑p (→p)*) ∪ (→p)* ,List of Words,((←p)*

mode(epsilon/1, [+]).
epsilon([[],[]]).
epsilon([[(right,Arg)|ArrowList]|Rest]) :-
        unfold(Arg,ArgList),
        match(ArgList,[ArrowList|Rest]).
epsilon(List) :-                  % reverse of previous clause
        appends(Rest,[Last],List), % Last = [PathBoxToCircle]
        appends(Arrowlist,[(left,Arg)],Last),
        appendp(Rest,[Arrowlist],NewList),
        unfold(Arg,ArgList),
        match(ArgList,NewList).
```

```
mode(break/5, [+,-,+,-,-]).
break(Whole,First,H,N,[H2|T2]) :-
        appends([H1|T1],[Word,H2|T2],Whole),
        lex(Word,Type),
        unfold(Type,Arrows),
        appends(ArrowList1,[H|N],Arrows),
        appendp([H1|T1],[ArrowList1],First).

mode(unfold/2, [+,-]).
unfold(F,[(up,F)]) :-
        atomic(F).
mode(atomic/1, [+]).
unfold((L,F,R),List) :-
        add_arrows(L,Lplus,left),
        add_arrows(R,Rplus,right),
        appendp(Lplus,[(up,F)|Rplus],List).

mode(add_arrows/3, [+,-,+]).
add_arrows([],[],_).
add_arrows([H1|T1],[(LorR,H1)|T2],LorR) :-
        add_arrows(T1,T2,LorR).

mode(appendp/3, [+,+,-]).
appendp([],K,K).
appendp([H|T],K,[H|L]) :-
        appendp(T,K,L).
mode(appends/3, [-,?,+]).
appends([],K,K).
appends([H|T],K,[H|L]) :-
        appends(T,K,L).
```

An example lexicon and query are:

```
mode(lex/2, [+,-]).
lex(marie,np).
lex(slaat,([np],s,[np])).
lex(loves,([np],s,[np])).
lex(de,([],np,[n])).
lex(vervelende,([],n,[([],pp,[([n],pp,[])])])).
lex(jongen,n).
lex(everyone,([],s,[([np],s,[])])).
lex(somebody,([([],s,[np])],s,[])).

?- recognize([everyone,loves,somebody]).
```

This is not a bottom-up algorithm as described in section 3.5, but a top-down one. In the predicate `match` we try to match a string of the form $((\overleftarrow{p})^* \overset{\uparrow}{p} (\overrightarrow{p})^*) \cup (\overrightarrow{p})^*$, the top, with a bottom of the form `[[PathCircleToBox],PathBoxToBox,[PathBoxToCircle]]`, i.e., $((\overleftarrow{p})^* \overset{\uparrow}{p} (\overrightarrow{p})^*) \cup (\overrightarrow{p})^*$,List of Words,$((\overleftarrow{p})^* \overset{\uparrow}{p} (\overrightarrow{p})^*) \cup (\overleftarrow{p})^*$

The bottom is a complicated way to represent two points in the chart. First we try to match the first element of the top with the first element of the body. If that's not possible we try to split the bottom in two parts and match the first element of the top with the first element of the second part of the bottom. In this case it must be possible to build an epsilon arc over the first element of the body. This is checked with the predicate `epsilon`. If we found a match for the first element of the top, we try to find a match for the rest.

The rest of the program should be clear.

7.2.2 Complexity Analysis

We apply the same method as in the previous analysis. The number of lexical items is $|Lex|$ and the length of the input is n. The length of the lexical items is constant.

```
mode(recognize/1, [+]).
recognize(L) :-
        append([[]|L],[[]],K),
        match([(up,s)],K).
```

$$\#\#(\{L,K\}) + \#\#(\{L,K\})$$

K and L are unique and of length n. Complexity is $\mathcal{O}(n)$

```
mode(match/2, [+,+]).
match([],L) :-
        epsilon(L).
```

$$\#\#(\{L\})$$

The second argument of *match* is of the form
$((\overleftarrow{p})^* \overset{\uparrow}{p} (\overrightarrow{p})^*) \cup (\overrightarrow{p})^*$,List of Words,$((\overleftarrow{p})^* \overset{\uparrow}{p} (\overrightarrow{p})^*) \cup (\overleftarrow{p})^*$.
The number of possible substitutions for $((\overleftarrow{p})^* \overset{\uparrow}{p} (\overrightarrow{p})^*) \cup (\overrightarrow{p})^*$ is $\mathcal{O}(|Lex|)$. The number of possible substitutions for $((\overleftarrow{p})^* \overset{\uparrow}{p} (\overrightarrow{p})^*) \cup (\overleftarrow{p})^*$ is also $\mathcal{O}(|Lex|)$.
Hence, $\#(\{L\}) = \mathcal{O}(|Lex|^2 \times n^2)$, $\#\#(\{L\}) = \mathcal{O}(|Lex|^2 \times n^3)$.

```
match([H|T1],[[H|T2]|T3]) :-
        match(T1,[T2|T3]).
```

$$\#\#(\{H,T1,T2,T3\})$$

The first argument of *match* is of the form $((\overleftarrow{p})^* \overset{\uparrow}{p} (\overrightarrow{p})^*) \cup (\overrightarrow{p})^*$. The number of possible substitutions for this argument is $\mathcal{O}(|Lex|)$. Hence, $\#(\{H,T1,T2,T3\}) = \mathcal{O}(|Lex| \times |Lex|^2 \times n^2)$ and $\#\#(\{H,T1,T2,T3\}) = \mathcal{O}(|Lex|^3 \times n^3)$.

```
match([H|T1],[[],[H|T2]]) :-
        match(T1,[[],T2]).
```

$\#\#(\{H, T1, T2\})$

$\mathcal{O}(|Lex|^2)$

```
match([H|T1],L) :-
        break(L,M,H,N,K),
        epsilon(M),
        match(T1,[N|K]).
```

$\#\#(\{H, T1, L, M, N, K\}) +$
$\#\#(\{H, T1, L, M, N, K\}) +$
$\#\#(\{H, T1, L, N, K\})$

Again, $\#([L]) = \mathcal{O}(|Lex|^2 \times n^2)$ and $\#([H, T1]) = \mathcal{O}(|Lex|)$. We have $n \times |Lex|$ splitting points. Resulting complexity is:
$\mathcal{O}(|Lex|^4 \times n^4) + \mathcal{O}(|Lex|^4 \times n^4) + \mathcal{O}(|Lex|^4 \times n^4) = \mathcal{O}(|Lex|^4 \times n^4)$

This is the most complex `match` clause. Therefore the complexity of all the match clauses is $\mathcal{O}(|Lex|^4 \times n^4)$.

```
mode(epsilon/1, [+]).
epsilon([[(right,Arg)|ArrowList]|Rest]) :-
        unfold(Arg,ArgList),
        match(ArgList,[ArrowList|Rest]).
```

$\#\#(\{Arg, ArrowList, Rest, ArgList\}) +$
$\#\#(\{Arg, ArrowList, Rest, ArgList\})$

The argument of `epsilon` is of the form
$((\overleftarrow{p})^* \overset{\uparrow}{p} (\overrightarrow{p})^*) \cup (\overrightarrow{p})^*$,List of Words,$((\overleftarrow{p})^* \overset{\uparrow}{p} (\overrightarrow{p})^*) \cup (\overleftarrow{p})^*$.
The number of possible substitutions for this argument is $\mathcal{O}(|Lex|^2 \times n^2)$. Every substitution for the argument uniquely determines the four variables in the rest of the clause, therefore $\#(\{Arg, ArrowList, Rest, ArgList\}) = \mathcal{O}(|Lex|^2 \times n^2)$ and
$\#\#(\{Arg, ArrowList, Rest, ArgList\}) = \mathcal{O}(|Lex|^2 \times n^3)$

```
epsilon(List) :-
        appends(Rest,[Last],List),
        appends(Arrowlist,[(left,Arg)],Last),
        appendp(Rest,[Arrowlist],NewList),
        unfold(Arg,ArgList),
        match(ArgList,NewList).
```

$\#\#(\{List, Rest, Last\}) +$
$\#\#(\{List, Rest, Last, Arrowlist, Arg\}) +$

$\#\#(\{List, Rest, Arrowlist, Arg, NewList\}) +$
$\#\#(\{List, Arg, NewList, ArgList\}) +$
$\#\#(\{List, NewList, ArgList\})$

The same holds as for the previous clause. Every substitution for *List* uniquely determines the substitution for the other variables in the clause (the clause is *deterministic*). The complexity of this clause is also $\mathcal{O}(|Lex|^2 \times n^3)$.

```
mode(break/5, [+,-,+,-,-]).
break(Whole,First,H,N,[H2|T2]) :-
        appends([H1|T1],[Word,H2|T2],Whole),
        lex(Word,Type),
        unfold(Type,Arrows),
        appends(ArrowList1,[H|N],Arrows),
        appendp([H1|T1],[ArrowList1],First).
```

$\#\#(\{Whole, H, H1, T1, Word, H2, T2\}) +$
$\#\#(\{Whole, H, H1, T1, Word, H2, T2, Type\}) +$
$\#\#(\{Whole, H, H1, T1, H2, T2, Type, Arrows\}) +$
$\#\#(\{Whole, H, H1, T1, H2, T2, Arrows, ArrowList1, N\}) +$
$\#\#(\{Whole, H, H1, T1, H2, T2, ArrowList1, N, First\})$

$\#(\{Whole, H\}) = \mathcal{O}(|Lex|^3 \times n^2)$. We first split `Whole` at some point (n possibilities). There are $|Lex|$ possibilities for $Type$.
$\#(\{Whole, H, H1, T1, Word, H2, T2, Type\})$ is $\mathcal{O}(|Lex|^4 \times n^3)$. If we split the arrow list, we have a constant amount of splitting points. Therefore,
$\#(\{Whole, H, H1, T1, H2, T2, Arrows, ArrowList1, N\}) = \mathcal{O}(|Lex|^4 \times n^3)$. Resulting complexity is $\mathcal{O}(|Lex|^4 \times n^4)$

```
mode(unfold/2, [+,-]).
unfold(F,[(up,F)]) :-
        atomic(F).
```

$\#\#(\{F\})$. `atomic` is a (built-in) predicate that we will not analyse. We assume it takes constant time. Complexity is $\mathcal{O}(|Lex|)$.

```
unfold((L,F,R),List) :-
        add_arrows(L,Lplus,left),
        add_arrows(R,Rplus,right),
        append(Lplus,[(up,F)|Rplus],List).
```

$\#\#(\{L, F, R, Lplus\}) +$
$\#\#(\{L, F, R, Lplus, Rplus\}) +$
$\#\#(\{L, F, R, Lplus, Rplus, List\})$

Complexity is $\mathcal{O}(|Lex|)$.

```
mode(add_arrows/3, [+,-,+]).
add_arrows([H1|T1],[(LorR,H1)|T2],LorR) :-
        add_arrows(T1,T2,LorR).
```

$\#\#(\{H1, T1, LorR, T2\})$

Complexity is $\mathcal{O}(|Lex|)$.

```
mode(appendp/3, [+,+,-]).
appendp([H|T],K,[H|L]) :-
        appendp(T,K,L).
```

$\#\#(\{H, T, K, L\})$. The complexity is lower than the complexity of the other clauses.

```
mode(appends/3, [-,?,+]).
appends([H|T],K,[H|L]) :-
        appends(T,K,L).
```

$\#\#([\langle\{K\}, 0\rangle, \langle\{H, L, T, K\}\rangle])$

Here we see a throughput variable for the first time. `appends` is called partially instantiated in `break`. The call is `appends(ArrowList1,[H2|N],Arrows)`. `Arrows` and `H2` are instantiated, `ArrowList1` and `N` are not. $\#\#(\{H, L\}) = |Lex|$. $\#\#([\langle\{K\}, 0\rangle]) = |Lex|$. $\#\#([\langle\{K\}, 0\rangle, \langle\{H, L, T, K\}\rangle]) = |Lex|^2$

For the other calls to `appends`, the complexity is lower than the complexity of the clause where they are called.
Summing the complexity of all the clauses gives us a bound of $\mathcal{O}(|Lex|^4 \times n^4)$.

Appendix A

An Implementation of the Earley Interpreter in Prolog

This appendix contains an implementation of the interpreter described in 6.1. An implementation in a pseudo imperative language has been given on page 96. Here we give in implementation in standard Prolog. The predicate *prove* replaces the pseudo program *main*. The first clause of *predict_or_complete* is the equivalent of lines 14-29 in program *main*. The second clause is the equivalent of 31-35. The third clause is the equivalent of the procedure *predict*. Because the stuff does not fit on one page we have split the program. Observe that we have explicitly stated where unification takes place by using the predicate `unify`. "Assignments" are marked by "=", i.e. "=" is no real unification.

```
:- dynamic ext_act_item/7, inactive_item/3, active_item/6,
           last_key/1, called/3.

prove :-
        clean,
        body(0,_,N,_,_,_,_),
        assert(called(0,success,0)),
        assert(active_item(0,0,N,0,[],[])),
        predict_or_complete(active_item(0,0,N,0,[],[])).
prove :-
        inactive_item(0, 0, success).
```

Figure A.1: Earley interpreter with explicit unification I

```
predict_or_complete(active_item(Cn,N,N,Key,HV,[])) :-
        head(Cn,N,Head,HVu),
        HVu = HV,
        assert(inactive_item(Cn,Key,Head)),
        ext_act_item(Key,Cn_mom,J1,M,Key_mom,HV_mom,RV_mom),
        Jp is J1 + 1,
        body(Cn_mom,Jp,M,HV_momu,Bi,RV_momu,RV_mom_new),
        HV_momu = HV_mom,
        RV_momu = RV_mom,
        unify(Head,Bi),
        assert_if(active_item(Cn_mom,Jp,M,Key_mom,HV_mom,RV_mom_new)),
        push(active_item(Cn_mom,Jp,M,Key_mom,HV_mom,RV_mom_new)),
        fail.

predict_or_complete(active_item(Cn,0,1,Key,HV,[])) :-
        body(Cn,1,1,_,true,[],[]),
        assert_if(active_item(Cn,1,1,Key,HV,[])),
        push(active_item(Cn,1,1,Key,HV,[])),fail.

predict_or_complete(active_item(Cn,J0,N,Key,HV,RV)) :-
        J0 < N,
        Jp is J0 + 1,
        body(Cn,Jp,N,HVu,Bi,RVu,RV_new),
        Bi \== true,
        HV = HVu,
        RV = RVu,
        head(Cn_son,M,Head1,HV_son),
        unify(Head1,Bi),
        freeze_term(HV_son,HV_son_nv),
        (called(Cn_son,HV_son_nv,Key_son) ->

        (assert_if(ext_act_item(Key_son,Cn,J0,N,Key,HV,RV)),
         inactive_item(Cn_son,Key_son,Solution),
         unify(Solution,Bi),
         assert_if(active_item(Cn,Jp,N,Key,HV,RV_new)),
         push(active_item(Cn,Jp,N,Key,HV,RV_new)));

        (add_called(Cn_son,HV_son_nv,Key_son),
         assert(ext_act_item(Key_son,Cn,J0,N,Key,HV,RV)),
         assert(active_item(Cn_son,0,M,Key_son,HV_son,[])),
         push(active_item(Cn_son,0,M,Key_son,HV_son,[])))),
        fail.

predict_or_complete(active_item(_,_,_,_,_,_)) :-
        stack(active_item(A,B,C,D,E,F)),
        retract(stack(active_item(A,B,C,D,E,F))),!,
        predict_or_complete(active_item(A,B,C,D,E,F)).
```

Figure A.2: Earley interpreter with explicit unification II

We have four database manipulation predicates. The first predicate adds something to a database if there is no alphabetic variant present as yet. If there is a variant, the predicate fails! The second predicate adds substitutions for the head variables to the `called` table. It assigns keys to all different table entries for faster lookup. The third predicate removes all tables. The fourth pushes item on the stack.

`freeze_term` and `variant` are auxiliary predicates.

```
assert_if(ext_act_item(Key_son,Cn,J,N,Key_mom,HV0,RV0)) :-
        \+ (ext_act_item(Key_son,Cn,J,N,Key_mom,HV1,RV1),
            variant((HV0,RV0),(HV1,RV1))),
        assert(ext_act_item(Key_son,Cn,J,N,Key_mom,HV0,RV0)).
assert_if(active_item(Cn,J,N,Key,HV0,RV0)) :-
        \+ (active_item(Cn,J,N,Key,HV1,RV1),
            variant((HV0,RV0),(HV1,RV1))),
        assert(active_item(Cn,J,N,Key,HV0,RV0)).

add_called(Cn,HV_son_nv,Y) :-
        last_key(X),
        Y is X + 1,
        retract(last_key(X)),
        assert(last_key(Y)),
        assert(called(Cn,HV_son_nv,Y)).

clean :-
        retractall(called(_,_,_)),
        retractall(last_key(_)),
        assert(last_key(0)),
        retractall(ext_act_item(_,_,_,_,_,_,_,_)),
        retractall(active_item(_,_,_,_,_,_)),
        retractall(inactive_item(_,_,_)),
        retractall(stack(_)).

push(A) :-
        asserta(stack(A)).

freeze_term(Term,Copy) :-
        copy_term(Term,Copy),
        numbervars(Copy,26,_).

variant(X,Y) :-
        \+ (\+ (numbervars(X,0,_),numbervars(Y,0,_),X == Y)).
```

Figure A.3: Auxiliary predicates for Earley interpreter

We can turn this interpreter into a breadth-first one that stops after finding the first solution by changing the last clause of `predict_or_complete` and the predicate `push`:

The breadth-first non-exhaustive interpreter is complete. It is often faster than the exhaustive search algorithm because it can stop earlier. If there is no solution it is slower because it checks (without success) very ofttten whether there is a solution in

```
predict_or_complete(active_item(_,_,_,_,_,_)) :-
        \+ inactive_item(0, 0, success), % stop after first solution
        stack(active_item(A,B,C,D,E,F)),
        retract(stack(active_item(A,B,C,D,E,F))),!,
        predict_or_complete(active_item(A,B,C,D,E,F)).

push(A) :-
        assertz(stack(A)). % breadth-first search
```

Figure A.4: Breadth-first non-exhaustive Earley interpreter

the table. The depth-first exhaustive interpreter is not able to find a proof when the search space is infinite. This occurs e.g in the following programs.

```
p(X) ::-
        [p((X,X))].
p(0) ::- [].

?- p(0).

pathplus(X,X,[X]) ::- [].
pathplus(X,Z,[X|Path]) ::-
        [edge(X,Y),
        pathplus(Y,Z,Path)].

edge(a,b) ::- [].
edge(b,c) ::- [].
edge(c,a) ::- [].
edge(c,d) ::- [].

?- pathplus(a,d,Path).
```

Figure A.5: Infinite search space

The first program is a trivial example. The second program is the path program with an extra argument. This argument should be instantiated to the path from `a` to `d` when the query `?- pathplus(a,d,Path).` has been proved. The depth-first exhaustive interpreter will get into an infinite loop. The breadth-first engine will find that the query is true. The method we presented only works when the search space (and the estimated time complexity) is finite. If the search space and estimated time complexity are infinite we cannot predict whether the prover will stop or not (this was already stated on page 73).

We mentioned on page 91 that there is a preprocessor that changes Prolog code into some internal format. The preprocessor is not given here. But it is not hard to imagine how it works. It “freezes” the variables and extracts the head variables and the relevant body variables. Then the variables are “melted” again. We proceed here with giving two example programs. We will show the original program and the

program in the internal format. The first program is an example of a DCG with left recursion. The second example illustrates that an occur check is sometimes necessary.

Program:

```
p([p|A],A).
p(B,C) :-
        p(B,A),
        q(A,C).

q([q|A],A).
```

Query:

```
?- p([p,q,q,q],[]).
```

(add imaginary clause 0: `success :- p([p,q,q,q],[]).`)
Translated into:

```
body(0, 1, 1, [], p([p,q,q,q],[]), [], []).
body(1, 1, 1, [A], true, [], []).
body(2, 1, 2, [B,C], p(B,A), [], [A]).
body(2, 2, 2, [B,C], q(A,C), [A], []).
body(3, 1, 1, [A], true, [], []).
head(0,1,success,[]).
head(1,1,p([p|A],A),[A]).
head(2,2,p(B,C),[B,C]).
head(3,1,q([q|A],A),[A]).
```

Figure A.6: Example program *DCG*

Program:

```
p(A,A).
```

Query:

```
?- p((Y,1),Y).
```

(add imaginary clause 0: `success :- p((Y,1),Y).`)
Translated into:

```
body(0, 1, 1, [], p((Y,1),Y), [], []).
body(1, 1, 1, [A], true, [], []).
head(0,1,success,[]).
head(1,1,p(A,A),[A]).
```

Figure A.7: Example program *occur check*

Appendix B

A Reduction of 3-SAT to ACSG Recognition

B.1 The Reduction

This appendix belongs to the reduction given on page 54. m is the number of variables in the 3-SAT formula. The variables i, j, k, l are necessary to define the grammar. u_i, u_j, u_k, u_l range over the variables in the 3-SAT formula. The rules of the form $A \to [B]C$ should be read as $B_1 A \to B_1 C$ (the B is the context). $i, j \in \{1, \ldots, m-1\}$ and $k, l \in \{1, \ldots, m\}$

In order to save space boolean variables are introduced.
$tv, tv', tv'', tv''' \in \{t, f\}$
$\tilde{tv}$ is the negated value of tv and is $\in \{t, f\}$

A. First initialize u_1:

$\mathrm{ini}\text{–}\mathrm{u}_1 tv \to \mathrm{ini}$

B. Pass the value of u_1 through the whole string:

$\neg \mathrm{u}_1 tv \to [\mathrm{ini}\text{–}\mathrm{u}_1 tv] \neg$
$\neg \mathrm{u}_1 tv \to [\mathrm{u}_{i+1} \mathrm{u}_1 tv] \neg$
$\neg \mathrm{u}_1 tv \to [tv' \mathrm{u}_1 tv] \neg$

$\mathrm{u}_{i+1} \mathrm{u}_1 tv \to [\mathrm{ini}\text{–}\mathrm{u}_1 tv] \mathrm{u}_{i+1}$
$\mathrm{u}_{i+1} \mathrm{u}_1 tv \to [\mathrm{u}_{j+1} \mathrm{u}_1 tv] \mathrm{u}_{i+1}$
$\mathrm{u}_{i+1} \mathrm{u}_1 tv \to [tv' \mathrm{u}_1 tv] \mathrm{u}_{i+1}$
$\mathrm{u}_{i+1} \mathrm{u}_1 tv \to [\neg \mathrm{u}_1 tv] \mathrm{u}_{i+1}$

C. u_1's are turned into true or false when its value is passed:

$tv\mathrm{u}_1tv \to [\mathrm{ini}\text{–}\mathrm{u}_1tv]\ \mathrm{u}_1$
$tv\mathrm{u}_1tv \to [\mathrm{u}_{j+1}\mathrm{u}_1tv]\ \mathrm{u}_1$
$tv\mathrm{u}_1tv \to [tv'\mathrm{u}_1tv]\ \mathrm{u}_1$

D. ¬'s disappear when the variables behind them are made true or false:

$\tilde{tv}\mathrm{u}_1tv \to \neg\mathrm{u}_1tv\ \ \mathrm{u}_1$

E. Initialize the next variable u_{i+1}:

$\mathrm{ini}\text{–}\mathrm{u}_{i+1}tv \to \mathrm{ini}\text{–}\mathrm{u}_itv'$

F. Pass the value through the formula across ¬'s:

$\neg\mathrm{u}_{j+1}tv \to [\mathrm{ini}\text{–}\mathrm{u}_{j+1}tv]\ \neg\mathrm{u}_jtv'$
$\neg\mathrm{u}_{j+1}tv \to [\mathrm{u}_k\mathrm{u}_{j+1}tv]\ \neg\mathrm{u}_jtv'\ (j < k-1)$
$\neg\mathrm{u}_{j+1}tv \to [tv''\mathrm{u}_{j+1}tv]\ \neg\mathrm{u}_jtv'$

G. Pass the value through the formula across t's and f's:

$tv''\mathrm{u}_{j+1}tv \to [\mathrm{ini}\text{–}\mathrm{u}_{j+1}tv]\ tv''\mathrm{u}_jtv'$
$tv''\mathrm{u}_{j+1}tv \to [\mathrm{u}_k\mathrm{u}_{j+1}tv]\ tv''\mathrm{u}_jtv'\ (j < k-1)$
$tv''\mathrm{u}_{j+1}tv \to [tv'''\mathrm{u}_{j+1}tv]\ tv''\mathrm{u}_jtv'$

H. Across u's which should not be made true or false:

$\mathrm{u}_l\mathrm{u}_{j+1}tv \to [\mathrm{ini}\text{–}\mathrm{u}_{j+1}tv]\ \mathrm{u}_l\mathrm{u}_jtv'\ (j < l-1)$
$\mathrm{u}_l\mathrm{u}_{j+1}tv \to [\mathrm{u}_k\mathrm{u}_{j+1}tv]\ \mathrm{u}_l\mathrm{u}_jtv'\ (j < l-1, j < k-1)$
$\mathrm{u}_l\mathrm{u}_{j+1}tv \to [tv''\mathrm{u}_{j+1}tv]\ \mathrm{u}_l\mathrm{u}_jtv'\ (j < l-1)$
$\mathrm{u}_l\mathrm{u}_{j+1}tv \to [\neg\mathrm{u}_{j+1}tv]\ \mathrm{u}_l\mathrm{u}_jtv'\ (j < l-1)$

I. These u's must be made true or false because the information about their initialization has arrived:

$tv\mathrm{u}_{i+1}tv \to [\mathrm{ini}\text{–}\mathrm{u}_{i+1}tv]\ \mathrm{u}_{i+1}\mathrm{u}_itv'$
$tv\mathrm{u}_{i+1}tv \to [\mathrm{u}_k\mathrm{u}_{i+1}tv]\ \mathrm{u}_{i+1}\mathrm{u}_itv'\ (i < k-1)$
$tv\mathrm{u}_{i+1}tv \to [tv''\mathrm{u}_{i+1}tv]\ \mathrm{u}_{i+1}\mathrm{u}_itv'$

J. ¬'s disappear again:

$\tilde{tv}\mathrm{u}_{i+1}tv \to \neg\mathrm{u}_{i+1}tv\ \ \mathrm{u}_{i+1}\mathrm{u}_itv'$

K. All values of u's have been passed now, start building an S:

$tv \rightarrow tv\mathrm{u}_m tv'$

$\mathrm{s} \rightarrow \mathrm{ini}\text{–}\mathrm{u}_m tv$

$\mathrm{s} \rightarrow \mathrm{s\,t\,t\,t}$
$\mathrm{s} \rightarrow \mathrm{s\,t\,t\,f}$
$\mathrm{s} \rightarrow \mathrm{s\,t\,f\,t}$
$\mathrm{s} \rightarrow \mathrm{s\,f\,t\,t}$
$\mathrm{s} \rightarrow \mathrm{s\,f\,f\,t}$
$\mathrm{s} \rightarrow \mathrm{s\,f\,t\,f}$
$\mathrm{s} \rightarrow \mathrm{s\,t\,f\,f}$

B.2 A Derivation

A possible derivation for the 3-SAT formula $(\mathrm{u}_2 \vee \neg\, \mathrm{u}_3 \vee \mathrm{u}_1)$. u_1 and u_2 are initialized as true. u_3 is initialized as false. The formula is changed into a cluster of three t's. Behind the string is indicated which rule is applied. The characters A, B, ... show from which group of rules the rule is taken.

ini	u_2	$\neg$	u_3	u_1	A	$\mathrm{ini}\text{–}\mathrm{u}_1\mathrm{t} \rightarrow \mathrm{ini}$
$\mathrm{ini}\text{–}\mathrm{u}_1\mathrm{t}$	u_2	$\neg$	u_3	u_1	B	$\mathrm{u}_2\mathrm{u}_1\mathrm{t} \rightarrow [\mathrm{ini}\text{–}\mathrm{u}_1\mathrm{t}]\ \mathrm{u}_2$
$\mathrm{ini}\text{–}\mathrm{u}_1\mathrm{t}$	$\mathrm{u}_2\mathrm{u}_1\mathrm{t}$	$\neg$	u_3	u_1	E	$\mathrm{ini}\text{–}\mathrm{u}_2\mathrm{t} \rightarrow \mathrm{ini}\text{–}\mathrm{u}_1\mathrm{t}$
$\mathrm{ini}\text{–}\mathrm{u}_2\mathrm{t}$	$\mathrm{u}_2\mathrm{u}_1\mathrm{t}$	$\neg$	u_3	u_1	B	$\neg\mathrm{u}_1\mathrm{t} \rightarrow [\mathrm{u}_2\mathrm{u}_1\mathrm{t}]\ \neg$
$\mathrm{ini}\text{–}\mathrm{u}_2\mathrm{t}$	$\mathrm{u}_2\mathrm{u}_1\mathrm{t}$	$\neg\mathrm{u}_1\mathrm{t}$	u_3	u_1	I	$\mathrm{tu}_2\mathrm{t} \rightarrow [\mathrm{ini}\text{–}\mathrm{u}_2\mathrm{t}]\ \mathrm{u}_2\mathrm{u}_1\mathrm{t}$
$\mathrm{ini}\text{–}\mathrm{u}_2\mathrm{t}$	$\mathrm{tu}_2\mathrm{t}$	$\neg\mathrm{u}_1\mathrm{t}$	u_3	u_1	E	$\mathrm{ini}\text{–}\mathrm{u}_3\mathrm{f} \rightarrow \mathrm{ini}\text{–}\mathrm{u}_2\mathrm{t}$
$\mathrm{ini}\text{–}\mathrm{u}_3\mathrm{f}$	$\mathrm{tu}_2\mathrm{t}$	$\neg\mathrm{u}_1\mathrm{t}$	u_3	u_1	B	$\mathrm{u}_3\mathrm{u}_1\mathrm{t} \rightarrow [\neg\mathrm{u}_1\mathrm{t}]\ \mathrm{u}_3$
$\mathrm{ini}\text{–}\mathrm{u}_3\mathrm{f}$	$\mathrm{tu}_2\mathrm{t}$	$\neg\mathrm{u}_1\mathrm{t}$	$\mathrm{u}_3\mathrm{u}_1\mathrm{t}$	u_1	F	$\neg\mathrm{u}_2\mathrm{t} \rightarrow [\mathrm{tu}_2\mathrm{t}]\ \neg\mathrm{u}_1\mathrm{t}$
$\mathrm{ini}\text{–}\mathrm{u}_3\mathrm{f}$	$\mathrm{tu}_2\mathrm{t}$	$\neg\mathrm{u}_2\mathrm{t}$	$\mathrm{u}_3\mathrm{u}_1\mathrm{t}$	u_1	G	$\mathrm{tu}_3\mathrm{f} \rightarrow [\mathrm{ini}\text{–}\mathrm{u}_3\mathrm{f}]\ \mathrm{tu}_2\mathrm{t}$
$\mathrm{ini}\text{–}\mathrm{u}_3\mathrm{f}$	$\mathrm{tu}_3\mathrm{f}$	$\neg\mathrm{u}_2\mathrm{t}$	$\mathrm{u}_3\mathrm{u}_1\mathrm{t}$	u_1	C	$\mathrm{tu}_1\mathrm{t} \rightarrow [\mathrm{u}_3\mathrm{u}_1\mathrm{t}]\ \mathrm{u}_1$
$\mathrm{ini}\text{–}\mathrm{u}_3\mathrm{f}$	$\mathrm{tu}_3\mathrm{f}$	$\neg\mathrm{u}_2\mathrm{t}$	$\mathrm{u}_3\mathrm{u}_1\mathrm{t}$	$\mathrm{tu}_1\mathrm{t}$	H	$\mathrm{u}_3\mathrm{u}_2\mathrm{t} \rightarrow [\neg\mathrm{u}_2\mathrm{t}]\ \mathrm{u}_3\mathrm{u}_1\mathrm{t}$
$\mathrm{ini}\text{–}\mathrm{u}_3\mathrm{f}$	$\mathrm{tu}_3\mathrm{f}$	$\neg\mathrm{u}_2\mathrm{t}$	$\mathrm{u}_3\mathrm{u}_2\mathrm{t}$	$\mathrm{tu}_1\mathrm{t}$	F	$\neg\mathrm{u}_3\mathrm{f} \rightarrow [\mathrm{tu}_3\mathrm{f}]\ \neg\mathrm{u}_2\mathrm{t}$
$\mathrm{ini}\text{–}\mathrm{u}_3\mathrm{f}$	$\mathrm{tu}_3\mathrm{f}$	$\neg\mathrm{u}_3\mathrm{f}$	$\mathrm{u}_3\mathrm{u}_2\mathrm{t}$	$\mathrm{tu}_1\mathrm{t}$	G	$\mathrm{tu}_2\mathrm{t} \rightarrow [\mathrm{u}_3\mathrm{u}_2\mathrm{t}]\ \mathrm{tu}_1\mathrm{t}$
$\mathrm{ini}\text{–}\mathrm{u}_3\mathrm{f}$	$\mathrm{tu}_3\mathrm{f}$	$\neg\mathrm{u}_3\mathrm{f}$	$\mathrm{u}_3\mathrm{u}_2\mathrm{t}$	$\mathrm{tu}_2\mathrm{t}$	J	$\mathrm{tu}_3\mathrm{f} \rightarrow \neg\mathrm{u}_3\mathrm{f}\ \mathrm{u}_3\mathrm{u}_2\mathrm{t}$
$\mathrm{ini}\text{–}\mathrm{u}_3\mathrm{f}$	$\mathrm{tu}_3\mathrm{f}$	$\mathrm{tu}_3\mathrm{f}$		$\mathrm{tu}_2\mathrm{t}$	G	$\mathrm{tu}_3\mathrm{f} \rightarrow [\mathrm{tu}_3\mathrm{f}]\ \mathrm{tu}_2\mathrm{t}$
$\mathrm{ini}\text{–}\mathrm{u}_3\mathrm{f}$	$\mathrm{tu}_3\mathrm{f}$	$\mathrm{tu}_3\mathrm{f}$		$\mathrm{tu}_3\mathrm{f}$	K	$\mathrm{t} \rightarrow \mathrm{tu}_3\mathrm{f}$
$\mathrm{ini}\text{–}\mathrm{u}_3\mathrm{f}$	t	$\mathrm{tu}_3\mathrm{f}$		$\mathrm{tu}_3\mathrm{f}$	K	$\mathrm{t} \rightarrow \mathrm{tu}_3\mathrm{f}$
$\mathrm{ini}\text{–}\mathrm{u}_3\mathrm{f}$	t	t		$\mathrm{tu}_3\mathrm{f}$	K	$\mathrm{t} \rightarrow \mathrm{tu}_3\mathrm{f}$
$\mathrm{ini}\text{–}\mathrm{u}_3\mathrm{f}$	t	t		t	K	$\mathrm{s} \rightarrow \mathrm{ini}\text{–}\mathrm{u}_3\mathrm{f}$
s	t	t		t	K	$\mathrm{s} \rightarrow \mathrm{s\,t\,t\,t}$
		s				

Bibliography

Aarts, Erik: 1991, Recognition for Acyclic Context-Sensitive Grammars is probably Polynomial for Fixed Grammars, *ITK Research Memo no. 8*, Tilburg University.

Aarts, Erik: 1992, Uniform Recognition for Acyclic Context-Sensitive Grammars is NP-complete, *Proceedings of COLING '92*, Nantes, pp. 1157–1161.

Aarts, Erik: 1994a, Parsing Second Order Lambek Grammar in Polynomial Time, *in* Paul Dekker and Martin Stokhof (eds), *Proceedings of the Ninth Amsterdam Colloquium*, Institute for Logic, Language and Computation, University of Amsterdam, pp. 35–45.

Aarts, Erik: 1994b, Proving Theorems of the Lambek calculus of order 2 in Polynomial Time, *Studia Logica* **53**(3), 373–387.

Aarts, Erik and Trautwein, Kees: 1996, Non-associative Lambek Categorial Grammar in Polynomial Time, *Mathematical Logic Quarterly* **42**(1), 000–000.

Abramson, H. and Dahl, V.: 1989, *Logic Grammars*, Springer, New York–Heidelberg–Berlin.

Apt, K. R. and Pellegrini, A.: 1994, On the occur-check free Prolog programs, *ACM Toplas* **16**(3), 687–726.

Barry, Guy: 1992, *Derivation and Structure in Categorial Grammar*, PhD thesis, University of Edinburgh.

Barton Jr., G. Edward, Berwick, Robert C. and Ristad, Eric Sven: 1987, *Computational Complexity and Natural Language*, MIT Press, Cambridge, MA.

Bunt, Harry: 1988, DPSG and its use in sentence generation from meaning representations, *in* M. Zock and G. Sabah (eds), *Advances in Natural Language Generation*, Pinter Publishers, London.

Buntrock, Gerhard: 1993, Growing Context-sensitive Languages and Automata, *Report no. 69*, University of Würzburg, Computer Science.

Buntrock, Gerhard and Loryś, Krzysztof: 1992, On growing context-sensitive languages, *Proceedings of 19th International Colloquium on Automata, Languages and Programming*, Vol. 623 of *Lecture Notes in Computer Science*, Springer, pp. 77–88.

Buszkowski, W.: 1986, Generative Capacity of the Nonassociative Lambek Calculus,

Bull. Pol. Acad. Scie. Math. **34**, 507–516.

Buszkowski, W.: 1990, On generative capacity of the Lambek calculus, *in* J. van Eijck (ed.), *JELIA '90: Logics in AI*, Amsterdam, pp. 139–152.

Buszkowski, W., Marciszewski, W. and van Benthem, Johan (eds): 1988, *Categorial Grammar*, Linguistic and Literary Studies in Eastern Europe (LLSEE), John Benjamins Publishing Company, Amsterdam–Philadelphia.

Chomsky, N.: 1959, A note on phrase-structure Grammars, *Information and Control* **2**, 137–167.

Cohen, J. M.: 1967, The equivalence of two concepts of categorial grammar, *Information and Control* **10**, 475–484.

Cormen, Thomas H., Leiserson, Charles E. and Rivest, Ronald L.: 1990, *Introduction to algorithms*, MIT Press.

Dahlhaus, Elias and Warmuth, Manfred K.: 1986, Membership for Growing Context-Sensitive Grammars is Polynomial, *Journal of Computer and System Sciences* **33**, 456–472.

Debray, Saumya K. and Lin, Nai-Wei: n.d., Cost Analysis of Logic Programs, Available as ftp://cs.arizona.edu/people/debray/papers/cost_analysis.ps.

Deransart, Pierre and Maluszynski, Jan: 1993, *A Grammatical View of Logic Programming*, MIT< Press, Cambridge, Mass.

Earley, Jay: 1970, An Efficient Context-Free Parsing Algorithm, *Communications of the ACM* **13**(2), 94–102.

Garey, Michael R. and Johnson, David S.: 1979, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, San Francisco, CA.

Gazdar, Gerald, Klein, Ewan, Pullum, Geoffrey and Sag, Ivan: 1985, *Generalized Phrase Structure Grammar*, Blackwell.

Hendriks, Herman: 1993, *Studied flexibility*, ILLC dissertation series 1993-5, University of Amsterdam, Amsterdam.

Hepple, Mark: 1991, Efficient Incremental Processing with Categorial Grammar, *Proc. of ACL-27*.

Janssen, Theo M.V.: 1991, On Properties of the Zielonka-Lambek Calculus, *in* Paul Dekker and Martin Stokhof (eds), *Proceedings of the Eighth Amsterdam Colloquium*, Institute for Logic, Language and Computation, University of Amsterdam, pp. 303–308.

Johnson, Mark: 1985, Parsing with discontinuous constituents, *Proceedings of the 23rd Ann. Meeting of the ACL*, pp. 127–132.

Johnson, Mark: 1988, *Attribute-Value Logic and the Theory of Grammar*, Vol. 16 of *CSLI Lecture Notes*, CSLI, Stanford.

Joshi, Aravind K., Levy, L.S. and Takahashi, A. M.: 1975, Tree adjunct grammars, *International Journal of Computer and Information Sciences*.

Joshi, Aravind K., Vijay-Shanker, K. and Weir, David: 1991, The Convergence of Mildly Context-Sensitive Grammar Formalisms, *in* P. Sells, S.M. Shieber and T. Wasow (eds), *Foundational Issues in Natural Language Processing*, MIT Press, Cambridge, MA, pp. 31–81.

Kandulski, Maciej: 1988, The non-associative Lambek calculus, *in* W. Buszkowski, W. Marciszewski and Johan van Benthem (eds), *Categorial Grammar*, Linguistic and Literary Studies in Eastern Europe (LLSEE), John Benjamins Publishing Company, Amsterdam–Philadelphia, pp. 141–151.

Kanovich, Max I.: 1991, The Horn fragment of linear logic is NP-complete, *ITLI prepublication series X-91-14*, University of Amsterdam, Amsterdam.

Kaplan, R. and Bresnan, J.: 1982, Lexical-Functional Grammar: a Formal System for Grammatical Representation, *in* J. Bresnan (ed.), *The Mental Representation of Grammatical Relations*, MIT Press, Cambridge, MA, pp. 173–281.

Karp, R. M.: 1972, Reducibility among combinatorial problems, *in* R. E. Miller and J. W. Thatcher (eds), *Complexity of Computer Computations*, Plenum Press, New York, pp. 85–103.

Kuroda, S.-Y.: 1964, Classes of Languages and Linear-Bounded Automata, *Information and Control* **7**, 207–223.

Lambek, J.: 1958, The Mathematics of Sentence Structure, *American Mathematical Monthly* **65**, 154–169.

Lewis, Harry R. and Papadimitriou, Christos H.: 1981, *Elements of the theory of computation*, Prentice-Hall, Englewood Cliffs, NJ.

Lin, Nai-Wei: 1993, *Automatic Complexity Analysis of Logic Programs*, PhD thesis, Department of Computer Science, University of Arizona. Available as ftp://cs.arizona.edu/caslog/naiwei93.tar.Z.

Lloyd, J. W.: 1987, *Foundations of Logic Programming*, Springer Verlag, New York–Heidelberg–Berlin.

Moortgat, Michael: 1988, *Categorial Investigations. Logical and Linguistic Aspects of the Lambek Calculus*, PhD thesis, Universiteit van Amsterdam.

Moortgat, Michael: 1995, Multimodal linguistic inference, to appear in Journal of Logic, Language and Information.

van Noord, Gertjan: 1993, *Reversibility in Natural Language Processing*, PhD thesis, Utrecht University.

Paterson, M.S. and Wegman, M. N.: 1978, Linear Unification, *Journal of Computer and System Sciences* **16**, 158–167.

Pentus, Mati: 1993, Lambek Grammars are Context Free, *IEEE Symposium on Logic in Computer Science*.

Pereira, F. and Warren, D.H.: 1983, Parsing as deduction, *Proceedings of the 21st Ann. Meeting of the ACL*, MIT, Cambridge, Mass., pp. 137–144.

Pereira, Fernando: 1981, Extraposition Grammars, *Computational Linguistics* **7**(4), 243–256.

Pereira, Fernando: 1993, Review of "The Logic of Typed Feature Structures" (Bob Carpenter), *Computational Linguistics* **19**(3), 544–552.

Pereira, Fernando C.N. and Shieber, Stuart M.: 1987, *Prolog and Natural Language Analysis*, Vol. 10 of *CSLI Lecture Notes*, Stanford.

Roorda, Dirk: 1991, *Resource Logics: Proof-theoretical Investigations*, PhD thesis, University of Amsterdam.

Salomaa, Arto: 1973, *Formal languages*, Academic Press.

Schabes, Yves and Joshi, Aravind K.: 1988, An Earley-type parsing algorithm for Tree Adjoining Grammars, *26th Annual Meeting of the Association for Computational Linguistics*, Buffalo, pp. 258–269.

Shapiro, Ehud Y.: 1984, Alternation and the Computational Complexity of Logic Programs, *Journal of Logic Programming* **1**, 19–33.

Sippu, Seppo and Soisalon-Soininen, Eljas: 1988, *Parsing Theory, Vol. 1: Languages and Parsing*, EATCS Monographs on Theoretical Computer Science, Springer Verlag.

Tamaki, H. and Sato, T.: 1986, OLDT resolution with tabulation, *Proc. of 3rd Int. Conf. on Logic Programming*, Springer-Verlag, Berlin, pp. 84–98.

Trautwein, C.: 1991, *Een parseeralgoritme voor de niet associatieve Lambek calculus (in dutch)*, Master's thesis, University of Amsterdam. [A Parsing Algorithm for the Non-Associative Lambek Calculus].

Vieille, L.: 1989, Recursive query processing: the power of logic, *Theoretical Computer Science* **69**, 1–53.

Warren, David H.D.: 1975, Earley deduction, unpublished note.

Warren, David S.: 1991, Computing the Well-Founded Semantics of Logic Programs, *Technical report 91/12*, Computer Science Department, SUNY at Stony Brook.

Warren, David S.: 1992, Memoing for Logic Programs, *Communications of the ACM* **35**(3), 94–111.

Winograd, Terry: 1983, *Language as a cognitive process: syntax*, Addison Wesley, Reading, MA.

Zielonka, W.: 1978, A direct proof of the equivalence of free categorial grammars and simple phrase structure grammars, *Studia Logica* **37**, 41–57.

Samenvatting

In dit proefschrift wordt de tijdscomplexiteit van drie problemen bekeken. De tijdscomplexiteit van een probleem is de relatie tussen de benodigde rekentijd om een probleem op een computer op te lossen en de omvang van het probleem. In deze relatie laten we constante factoren weg. Stel dat we een object en een ongesorteerde lijst hebben, en dat we willen weten of het object voorkomt in de lijst. De omvang van het probleem is de lengte van de lijst. Die noemen we n. Als we de lijst langslopen en elk object in de lijst vergelijken met het object dat we zoeken is de benodigde rekentijd in het slechtste geval $\mathcal{O}(n)$ (orde n). Als de relatie tussen rekentijd en omvang van een probleem een polynomiale functie is ($\mathcal{O}(n^k)$ met k vast), zeggen we dat het probleem in P (Polynomiale tijd) zit.

Vaak is een probleem moeilijk op te lossen, maar is het controleren van de oplossing eenvoudig. Stel we hebben een groot getal dat het product is van twee grote priemgetallen en dat het probleem is: "vind de twee priemfactoren". Een zeer eenvoudig algoritme is het volgende: probeer alle kleinere getallen als deler van het getal dat ontbonden moet worden. Als het getal n cijfers heeft, moeten we $\mathcal{O}(10^n)$ getallen proberen als priemfactor. Dit is duidelijk niet polynomiale tijd, maar exponentiële. Er zijn overigens veel betere algoritmes bekend, maar die werken ook in exponentiële tijd. Het controleren van een oplossing kan wel in polynomiale tijd: als we twee getallen hebben hoeven we alleen maar te vermenigvuldigen om te zien of het product inderdaad het getal is dat ontbonden moest worden. De klasse van problemen met de eigenschap dat oplossingen gemakkelijk gecontroleerd kunnen worden heet NP (nondeterministische polynomiale tijd). Problemen in NP waarvan vermoed wordt dat ze niet in P zitten heten NP-compleet. Ontbinden in priemfactoren is NP-compleet.

Het eerste deel van dit proefschrift beschrijft de tijdscomplexiteit van problemen in twee gebieden: de categoriale grammatica's en de acyclische context-gevoelige grammatica's. Categoriale grammatica's bestaan uit een lexicon waarin types worden toegekend aan lexicale elementen en uit een sequenten calculus waarin geredeneerd kan worden over afleidbaarheid van types. Een type A is afleidbaar uit een rijtje typen Γ als de sequent $\Gamma \to \mathsf{A}$ afleidbaar is in de sequenten calculus. We kunnen een categoriale taal definiëren als die verzameling strings waarvoor geldt dat ze via het lexicon op een rijtje types afgebeeld kunnen worden en dat uit dit rijtje het type s (of een ander "starttype") afleidbaar is in de sequenten calculus. Het probleem dat we bekijken is

dat we van een bepaalde string willen weten of die in de categoriale taal zit. Het is niet bekend of dit probleem, bij gebruik van de standaard Lambek calculus, in P zit of juist NP-compleet is. Dit proefschrift laat zien dat voor bepaalde fragmenten het probleem in P zit. Die fragmenten zijn de grammatica's waarbij de niet-associatieve Lambek calculus en de tweede orde Lambek calculus worden gebruikt (met tweede orde wordt bedoeld dat de nestingsdiepte van types hooguit twee is).
Acyclische context-gevoelige grammatica's zijn herschrijf grammatica's die boomstructuren met kruisende takken kunnen genereren. Dit gebeurt door indices toe te kennen aan de context elementen in de herschrijfregel. We hebben gekeken naar het probleem of een bepaalde string in de gegenereerde taal zit. Als we gewone context-gevoelige grammatica's gebruiken is dit een heel moeilijk probleem, moeilijker dan de NP-complete problemen. Daarom eisen we dat de grammatica acyclisch is. We laten zien dat het probleem in P zit als we alleen de invoerzinnen variëren en de grammatica vast houden. Als de grammatica ook mag variëren (en dus een rol speelt in "de omvang van het probleem"), dan wordt het probleem NP-compleet.
Het tweede deel van dit proefschrift gaat over Prolog programma's. Prolog is een zeer eenvoudige, krachtige programmeertaal. Het grootste verschil met andere programmeertalen is dat het mogelijk is om niet-deterministische programma's te coderen. Omdat computers deterministische machines zijn moeten Prolog programma's op een bepaalde manier omgezet worden naar deterministische programma's. De standaard manier om dit te doen is eerst-in-de-diepte (*depth-first*) met terugkrabbelen (*backtracking*). Als er een keuze gemaakt moet worden, wordt de eerste optie genomen. Als blijkt dat deze keuze op geen enkele manier tot resultaat leidt wordt de volgende optie geprobeerd, enzovoort. In dit proefschrift wordt naar een alternatieve zoekstrategie (OLDT resolutie) gekeken. In deze strategie wordt bijgehouden welke alternatieven geprobeerd zijn en met welk resultaat. Deze strategie voorkomt dat een bepaalde deelberekening twee keer wordt uitgevoerd. In de standaard methode kan de zoekruimte als een boom gerepresenteerd worden. Twee knopen in de boom kunnen best hetzelfde deelprobleem representeren. Bij OLDT resolutie representeren alle knopen in de zoekruimte per definitie verschillende deelproblemen; de zoekruimte is dan ook geen boom meer maar een graaf. Het idee dat in dit proefschrift uitgewerkt is, is dat de omvang van de zoekruimte een maat is voor de rekentijd die door een Prolog programma wordt gebruikt, daar elke tak in de zoekruimte maar één keer bewandeld wordt. In het tweede deel van het proefschrift wordt een methode gegeven om van een Prolog programma de benodigde rekentijd te schatten onder de assumptie dat OLDT-resolutie wordt gebruikt bij de executie van het programma.
Tenslotte worden de delen 1 en 2 gecombineerd. We geven Prolog programma's die uitrekenen of een string in een categoriale taal zit (voor de beide fragmenten). Vervolgens laten we zien dat de rekentijd, die die Prolog programma's nodig hebben, polynomiaal is in de lengte van de zin en in de lengte van het lexicon.

Summary

In this dissertation the time complexity of three problems is considered. The time complexity of a problem is the relation between the time a computer needs to solve a problem and the size of that problem. In this relation, we leave out constant factors. Suppose we have an object and an unsorted list, and that we want to know whether the object occurs in the list or not. The size of the problem is the length of the list. We call this length n. When we walk through the list and compare each object in the list with the object that we are looking for, the time needed is $\mathcal{O}(n)$ (order n) in the worst case. When the relation between the computing time and the size of the problem is a polynomial function ($\mathcal{O}(n^k)$ with k fixed), we say that the problem is in P (Polynomial time).

Often a problem is hard to solve, but checking a solution is simple. Suppose we have a big number which is the product of two big prime numbers and that we are asked to find the two prime factors. A very simple algorithm is the following: try all smaller numbers as a divisor of the number we have to factor. If the number has n digits, we have to try $\mathcal{O}(10^n)$ numbers as a prime factor. Apparently, this is not polynomial time, but exponential. By the way, there are much better algorithms, but these algorithms are not polynomial time either. Checking a solution can be done in polynomial time, however, if we have two numbers, we only have to multiply them in order to see whether the product is indeed the number we had to factor. The class of problems with the property that solutions can be checked easily, is called NP (Nondeterministic Polynomial time). Problems in NP that are conjectured to be not in P are called NP-complete. Factoring a number in prime numbers is NP-complete.

The first part of this thesis describes the time complexity of problems in two fields: categorial grammar and acyclic context-sensitive grammar. Categorial grammars consist of a lexicon, in which types are assigned to lexical elements, and a sequent calculus in which we can reason about derivability of types. A type A is derivable from a sequence of types Γ if the sequent $\Gamma \to \mathsf{A}$ is derivable in the sequent calculus. We can define a categorial language as the set of strings which can be mapped, via the lexicon, onto a sequence of types, for which it holds that the start type s is derivable from the sequence. The problem that we consider is the following: for a given string, we want

to know whether it is in the categorial language or not. It is not known whether this problem, when the standard Lambek calculus is used, is in P or NP-complete. This dissertation shows that for certain fragments the problem is in P. Those fragments are the grammars that use the non-associative Lambek calculus and the second order calculus (with second order we mean that the nesting depth is limited to two).
Acyclic context-sensitive grammars are rewrite grammars that generate tree structures with crossing branches. We assign indices to context elements in the rewrite rule. We look at the problem whether a certain string is in the language generated. When we use normal context-sensitive grammars, this is a very hard problem, harder than NP-complete problems. Therefore we use acyclic grammars. We show that the problem, for these grammars, is in P if we vary the sentences only, and keep the grammar fixed. If the grammar may change as well, (and plays a role in the "size of the problem"), then the problem becomes NP-complete.
The second part of this dissertation is about Prolog programs. Prolog is a very simple, powerful programming language. The big difference with other programming languages is that it is possible to code non-deterministic programs. But computers are deterministic machines, so the Prolog program must be converted to a deterministic program in some way. The standard way to do this is depth-first with backtracking. If a choice must be made, the first option is taken. When it turns out that this option does not lead to any result, the next option is tried, and so on. In this dissertation we consider an alternative search strategy (OLDT resolution). In this strategy, one memoes which alternatives are tried and what the result was. This strategy prevents that some subcomputation is done twice. In the standard search, the search space can be represented as a tree. It is very well possible that two nodes in the search space represent the same subproblem. In OLDT resolution, all nodes represent different subproblems by definition; the search space is not a tree anymore, but a graph. The idea that has been elaborated in this dissertation is that the size of the search space is an estimate for the time a Prolog program needs, because every branch in the search space is visited only once. In the second part of this dissertation a method is given to estimate the time a Prolog program needs under the assumption that OLDT resolution is used in the execution of the program.
Finally we combine parts 1 and 2. We give Prolog programs, that compute whether some string is in a categorial language (for both fragments). Then we show that the time that those Prolog programs need, is polynomial in the length of the sentence and in the length of the lexicon.

Curriculum Vitae

Erik Aarts was born on 25 May 1965 in Berlicum. In 1983 he obtained the VWO certificate at "Gymnasium Bernrode" in Heeswijk-Dinther. In 1984 he passed the first-year examination of Computer Science at the University of Technology in Eindhoven. He further pursued his academic career at the Department of Philosophy and Social Sciences at the same university. In 1989 he graduated from the University of Technology in "Technology and Society".
From 1989 until 1991 he worked at Tilburg University at the Institute for Language Technology and Artificial Intelligence as a researcher. From 1991 until 1995 he worked for the Netherlands Organization for Scientific Research (NWO). He was involved in a joint project from various universities in the Netherlands called 'Structural and Semantic Parallels in Natural Languages and Programming Languages'. During this project he worked both at the Institute for Language and Speech at Utrecht University and at the Department of Mathematics and Computer Science at the University of Amsterdam.
He is married to Hilde van der Togt and has two daughters, Els and Pleun.

Titles in the ILLC Dissertation Series:

ILLC DS-1993-1: **Paul Dekker**
Transsentential Meditations; Ups and downs in dynamic semantics

ILLC DS-1993-2: **Harry Buhrman**
Resource Bounded Reductions

ILLC DS-1993-3: **Rineke Verbrugge**
Efficient Metamathematics

ILLC DS-1993-4: **Maarten de Rijke**
Extending Modal Logic

ILLC DS-1993-5: **Herman Hendriks**
Studied Flexibility

ILLC DS-1993-6: **John Tromp**
Aspects of Algorithms and Complexity

ILLC DS-1994-1: **Harold Schellinx**
The Noble Art of Linear Decorating

ILLC DS-1994-2: **Jan Willem Cornelis Koorn**
Generating Uniform User-Interfaces for Interactive Programming Environments

ILLC DS-1994-3: **Nicoline Johanna Drost**
Process Theory and Equation Solving

ILLC DS-1994-4: **Jan Jaspars**
Calculi for Constructive Communication, a Study of the Dynamics of Partial States

ILLC DS-1994-5: **Arie van Deursen**
Executable Language Definitions, Case Studies and Origin Tracking Techniques

ILLC DS-1994-6: **Domenico Zambella**
Chapters on Bounded Arithmetic & on Provability Logic

ILLC DS-1994-7: **V. Yu. Shavrukov**
Adventures in Diagonalizable Algebras

ILLC DS-1994-8: **Makoto Kanazawa**
Learnable Classes of Categorial Grammars

ILLC DS-1994-9: **Wan Fokkink**
Clocks, Trees and Stars in Process Theory

ILLC DS-1994-10: **Zhisheng Huang**
Logics for Agents with Bounded Rationality

ILLC DS-1995-1: **Jacob Brunekreef**
On Modular Algebraic Prototol Specification

ILLC DS-1995-2: **Andreja Prijatelj**
Investigating Bounded Contraction

ILLC DS-1995-3: **Maarten Marx**
Algebraic Relativization and Arrow Logic

ILLC DS-1995-4: **Dejuan Wang**
Study on the Formal Semantics of Pictures

ILLC DS-1995-5: **Frank Tip**
Generation of Program Analysis Tools

ILLC DS-1995-6: **Jos van Wamel**
Verification Techniques for Elementary Data Types and Retransmission Protocols

ILLC DS-1995-7: **Sandro Etalle**
Transformation and Analysis of (Constraint) Logic Programs

ILLC DS-1995-8: **Natasha Kurtonina**
Frames and Labels. A Modal Analysis of Categorial Inference

ILLC DS-1995-9: **G.J. Veltink**
Tools for PSF

ILLC DS-1995-10: **Giovanna Ceparello**
(to be announced)

ILLC DS-1995-11: **W.P.M. Meyer Viol**
Instantial Logic. An Investigation into Reasoning with Instances

ILLC DS-1995-12: **Szabolcs Mikulás**
Taming Logics

ILLC DS-1995-13: **Marianne Kalsbeek**
Metalogics for Logic Programming

ILLC DS-1995-14: **Rens Bod**
Enriching Linguistics with Statistics: Performance Models of Natural Language

ILLC DS-1995-15: **Marten Trautwein**
Computational Pitfalls in Tractable Grammatical Formalisms

ILLC DS-1995-16: **Sophie Fischer**
The Solution Sets of Local Search Problems

ILLC DS-1995-17: **Michiel Leezenberg**
Contexts of Metaphor

ILLC DS-1995-18: **Willem Groeneveld**
Logical Investigations into Dynamic Semantics

ILLC DS-1995-19: **Erik Aarts**
Investigations in Logic, Language and Computation