Diplomarbeit

Erweiterung des EMILE-Verfahrens zum induktiven Lernen von kontext-freien Grammatiken für natürliche Sprache.

Extension of the EMILE algorithm for inductive learning of context-free grammars for natural languages.

Erik Dörnenburg

Diplomarbeit

Erweiterung des EMILE-Verfahrens zum induktiven Lernen von kontext-freien Grammatiken für natürliche Sprache.

Extension of the EMILE algorithm for inductive learning of context-free grammars for natural languages.

Erik Dörnenburg

Dortmund, Oktober 1997

Betreuer

Prof. Dr. Katharina Morik

Thorsten Joachims

Universität Dortmund

Fachbereich Informatik

Contents

Preface

Most of the customary methods for knowledge discovery are not directly applicable to free-text entries in databases. These methods require a formal and rigid structure in the input against which passages of texts in natural languages with their ambiguity and richness of form appear to be unstructured, almost arbitrary. It is, however, also obvious that natural languages do adhere to certain rules and linguistic research has revealed clear principles, structure and functional dependencies on all levels of language, ranging from word formation to discourse. For a successful application of knowledge discovery techniques to free-texts it therefore seems to be necessary to transform the structure inherent in these texts into a more accessible, explicit form. This can be achieved in several steps starting with a syntactic analysis.

It is the goal of this project to devise a procedure that given short texts written by different authors learns a grammar that can be utilised to process other texts written by same or even different authors. The result of processing unknown input in the form of sentences should be a structural description of each sentence with categories that were established automatically. For the intended area of application, i.e. information extraction and data-mining, the algorithm must also be suited to handle large amounts of data efficiently.

Pieter Adriaans has carried out extensive work in the area of language learning. His research was motivated differently as it was his foremost interest to establish a model of dialogical language learning, to examine if and under which circumstances natural languages can be learnt and also to provide a formal proof for his findings. In the course of his research he developed several learning algorithms and a later version of his Emile algorithm seems to be suited to the task outlined above. In this thesis I will investigate whether the Emile algorithm with its theoretical background can indeed be used successfully in a very practical situation. There are, of course, a variety of different approaches to the problem and my concentration on the Emile algorithm should not be misunderstood to imply that only the Emile algorithm can be used to tackle the problem. This thesis is merely an exploration of one of the possible solutions.

In the first chapter I will present an overview of natural language processing in general and theories of grammar in particular. Some of these theories have a long tradition in computational models of language and I assume that the reader will be somewhat familiar with them. Nevertheless I present a brief overview in order to establish a consistent terminology and to explicate certain points which might be open to interpretation otherwise. A special emphasis is placed on those properties that are relevant in the envisioned area of application. I will also present an alternative theory of grammar, mainly because Adriaans' research is based on it. In chapter 2 I will introduce some concepts from machine learning theory and I will determine where Adriaans' approach is located in the field. I will also present other approaches to the problem of language/grammar learning, some of which can be understood as precursors of Adriaans' approach. Finally, I will describe in some detail the theoretical framework of the Emile algorithm. It might seem that chapters 1 and 2 are overly extensive in relation to the rest of the thesis but I believe that a sound understanding of the complete background is necessary for a proper evaluation of any approach to the problem. If certain aspects are left undiscussed, research might be directed in the wrong direction or false hope might be found in approaches

that are doomed right from the start. Also, the field of language/grammar learning seems to have a similar history as the area of connectionist computing, i.e. after a devastating criticism research activity was extremely low for an extended period of time before a realistic reassessment of the complexity issues involved resulted in a second spring. In the last couple of years a number of novel approaches to the problem of language/grammar learning have been explored but there are few works relating them to each other. Thus, chapter 2 can also be seen as my modest attempt to present an overview and comparison of several recent approaches to the problem.

In chapter 3 I will present the specific version of the Emile algorithm developed by Pieter Adriaans and Arno Knobbe which I believe might be suitable to solve the problem of learning a grammar for the purpose of structuring large amounts of text such that customary knowledge-discovery techniques can be applied to these texts. I will also describe a concrete implementation of the algorithm in an object-oriented environment and will present results from some experiments. These results are not extensive and systematic enough to provide an irrefutable characterization or an accurate quantitative analysis of the algorithm but they do indicate certain general problems that will inevitably occur when the algorithm is used for practical purposes. In chapters 4 and 5 I will explore various approaches to solve these problems and provide an evaluation of these approaches in terms of the experiments from chapter 3.

At this point I would like to take the opportunity to thank those people who have made this thesis possible. My supervisors, Katharina Morik and Thorsten Joachims for their valuable advice and guidance, my partner, Louise Drought for her patience throughout the period of writing and her constant help to make my English more understandable, my friend Moritz Thomas who ensured that I described what I meant, Pieter Adriaans for clarifying certain details and, of course, for providing the initial stimulus for my research. I would also like to thank the company Object Factory for providing software and especially hardware to carry out the experiments. Above all I would like to thank my parents for their constant support during my course of studies over the last few years.

Dortmund, October 1997

Erik Dörnenburg

Chapter

1 Natural language processing

In this chapter I will first present an overview of the motivation for natural language processing (NLP) and the grammar theory underlying the majority of language processing systems. This is important in so far as Adriaans' work, even though based on a different approach, can be related to the customary understanding of language and the practical results which I am interested in can be described in terms of this grammar theory. I will outline certain options and explain my choices. In section 1.3 I will present some ideas from a stochastic approach to language which I will incorporate into my work. In section 1.4 I will turn to the language processing paradigm under which Adriaans is working. I will give a brief introduction of the grammar theory involved and compare it to the standard approach.

1.1 Introduction

1.1.1 Motivation

There are, like in many other areas of AI, different motivations for research in NLP systems. The 'theoretical approach' is characterised by an interest to validate linguistic and psycholinguistics theories by means of an operational model, i.e. a model that is expressed in terms of a formalism suitable to be processed directly by a computer. This approach has its roots in the spirit of the Conference at Dartmouth College in 1956, which is generally considered the constituting meeting for research in artificial intelligence. Marvin Minsky noted that theories about mental processes had become too complex to be investigated without the help of an operational model. A successful NLP system within this approach would not only process language but it would process it in a way in which resembles the way that people 'process' languages and a system that learns a language would explain language acquisition. Due to its proximity in goals and methods, this approach is often referred to as computational linguistics.

Located on the other side of the spectrum is the 'engineering approach.' Here the sole interest is in practical applications of NLP systems, for example to improve human-computer interaction and information retrieval or to perform automated translations from a natural language into another one. No claim is made regarding a resemblance between the components of the NLP system, for example the grammar or the knowledge representation, and the structure of the brain. It is of no interest whether the system actually understands the language—if there was a widely accepted definition of what 'to understand a text' means. It is this approach to NLP that I will pursue in this thesis.

1.1.2 Levels of language description

As natural languages are complex systems they involve an abundance of structures and regularities. Consequently, a couple of subdisciplines of linguistics evolved, each concerning itself with a different aspect of language. Phonology is the study of the sounds of a language, the rules governing the realization of language in actual speech and vice versa. Morphology and

syntax are concerned with the structural description of basic language units, words and sentences respectively. Semantics captures the meaning of units of language. Regarding words it can be seen as an intermediary between the sign, the letters T-R-E-E for example, and the referent, an actual tree which can be physically experienced. Similar relations can be conceived between sentences and actions for example. Pragmatics provides theories about relations between smaller language units, about discourse structure, strategies employed by the participants in a dialogue etc. It should be noted that the there is an intricate network of relationships between these levels of description when it comes to actual language use. In the extreme, a pragmatic goal can be realized with phonological means.[1]

I will focus on learning the grammar or syntax of the language even though there is obviously more to a language than the wellformedness of a sentence. Natural languages are primarily a means to communicate and as such the semantics and pragmatics are at least as important as syntax. In fact, in most situations it is of lesser importance that an utterance was grammatical than that its meaning was communicated without loss of information. It appears that the whole purpose of a syntax is to ensure that the intended meaning is communicated with as little loss of information as possible. For example, consider a language without a syntax. In such a language it would not be clear from the sequence of words 'the cat chases the mouse' who is chasing who because there is no means to assign the words to the concepts of subject and object.[2]

The key to understanding the decision to concentrate on syntax lies in the engineering approach. It is my goal to transform a sequence of words into a structured representation which in turn can be utilized by other modules to implement semantic representations. These representations can be tailored to certain applications such as information extraction (IE) and data mining and will usually be quite different from linguistic and philosophic theories of meaning. I am aware that with this approach certain problems arise as I am transferring the separation of the levels of language description into a system that processes language. Consider the following sentences:

(1) I bought a plant with Mary

(2) I saw the man with a telescope.

Both sentences illustrate a phenomenon known as PP-attachment ambiguity. In both of them the prepositional-phrase could be attached to either the object or the whole verb-phrase. In sentence (1) semantics, i.e. the knowledge that Mary cannot be a part of a plant, rules out one possible reading. But in sentence (2) the ambiguity can only be resolved with the help of the context, with pragmatics. It seems, however, acceptable and even necessary to make certain sacrifices to implement an NLP system within the engineering approach.

1.1.3 Aspects of grammar formalisms

I will consider a grammar formalism to be a formalism to represent knowledge about language in general and syntax in particular. As such it should be set apart from a linguistic theory but in practice this distinction is often blurred and grammar formalisms are also considered to be theories for the domain of languages. This is especially true for the theoretical approach.

1 An example for this is the use of certain distinguished pronunciations to signal solidarity in a group.

2 Sometimes subject and object are even defined in terms of syntactical constructions.And sometimes this assignment can be made with the help of morphological instead of syntactical devices.

Grammar formalisms can be characterized on the basis of three independent aspects.

- The expressiveness of the formalism, i.e. the amount of phenomena within the targeted language the formalism can account for.
- The aptness for an algorithmic implementation. This includes the complexity in terms of time and space that potential implementations are bound to.
- The extent to which the formalism is linguistically motivated, i.e. the proximity between formal and conceptual properties of the formalism and linguistic theories.

The assessment of a grammar formalism according to these criteria greatly depends on the intended application. For work within the theoretical approach it is important that the formalism is linguistically motivated while an in engineering approach this would be traded in for ease of implementation.

1.2 The standard approach to NLP

1.2.1 The classical linguistic paradigm

Regardless of the motivation, most research in NLP is carried out in a structuralist linguistic paradigm established by Ferdinand de Saussure and, especially, Noam Chomsky. With the theoretical approach in mind it is also maintained that findings in one discipline are beneficial to the other. De Saussure's and Chomsky's approach to linguistics is centred around the following three assumptions that made it appealing to computer scientists.

The 'arbitrariness assumption' detaches the sign from its referent. Ultimately, it postulates that one can work with words without having to worry about their meaning. Under this assumption linguists developed theories of syntax that were void of semantics. In fact, a lot of early research in NLP was done with the assumption that the knowledge necessary for semantic processing would be provided by other disciplines of AI and could be incorporated when it would become available. I have already outlined related problems in paragraph 1.1.2.

The 'ideal speaker-listener assumption' suggests that it is possible and even necessary to establish different models for the knowledge a person has about language and for actual utterances in a language. Chomsky referred to these modes of language as competence and performance and stated clearly that he would concern himself with models for competence. While it is a methodologically valid procedure to idealize the subject of study, it caused problems when researchers built systems that were to deal with actual utterances using theories that were meant to explain competence.

The 'snapshot assumption' divides linguistics into a diachronic and a synchronic branch. It is assumed that a model of a language as observed at certain point in time can be established without considering the language's historic antecedents. This is clearly another idealisation that while limiting expenditure makes certain phenomena inexplainable.

Chomsky noted, obviously with the arbitrariness assumption in mind, that the mastery of syntax would be a necessary prerequisite to tackling semantics and pragmatics. I would, in line with advocates of a functional theories of linguistics,[3] argue that for a sound linguistic theory or

research within the theoretical approach this course of research is hazardous because it is built on too many assumptions that obstruct the view of language in its primary function: as a means of communication, a sociocultural activity. Further, the foundations of Chomsky's work have proven extremely useful in various areas of computer science, ranging from processing of natural and formal languages to the analysis of all sorts of structured data, but later refinements like Government and Binding theory[4] are relatively complex and therefore not really pervading NLP research.[5] Despite all this criticism, I am aware that contemplation of syntax alone can yield interesting results from a practical perspective and, as a matter of fact, the algorithm that I present in this thesis owes its simplicity to the concentration on syntactical phenomena. It does, however, also show that structure is not completely arbitrary as it derives semantic knowledge from syntactic structures.

The Emile algorithm in its theoretical incarnation in Adriaans work avoids the pitfalls associated with the classical linguistic paradigm as I will explain in section 2.3.

1.2.2 The Chomsky hierarchy of grammars

According to Chomsky a grammar is a system of rules with which all words[6] of a language can be generated or, formally, a four-tuple $<T, V, S, R>$ with:

- T, the finite alphabet over which the language is defined.
- V, a finite set of variables, with $V \cap T = \emptyset$.
- $S \in V$, the start symbol.
- R, a finite set of rules (or productions). A rule is a pair $l \rightarrow r$ with $l \in (V \cup T)^+$ and $r \in (V \cup T)^*$ and the following semantics: If a word w can be subdivided into abc such that $a, c \in (V \cup T)^*$ and $b \in (V \cup T)^+$ it can be replaced by $w' = adc$ iff R contains a rule $b \rightarrow d$.

The notation $w \rightarrow^* w''$ will be used if w'' can be derived from w in a finite number of steps. With this notation the language $L(G)$ generated by the grammar G is simply the set of all words $w \in T^*$, for which $S \rightarrow^* w$.

The Chomsky hierarchy is a set of four grammar classes that impose increasing restrictions on the types of rules allowed. Each grammar class is a proper subset of the foregoing one.

- Grammars without further restrictions are called Chomsky-0 grammars or general rewriting systems.
- Grammars in which all productions are of the form $axb \rightarrow ayb$ with $a, b \in V^*$, $x \in V$ and $y \in (V \cup T)^*$ or $S \rightarrow \varepsilon$ are called context-sensitive grammars or, abbreviated, CSGs.
- Grammars in which all productions are of the form $A \rightarrow w$ with $A \in V$ and $w \in (V \cup T)^*$ are called context-free grammars or, abbreviated, CFGs.

---

3 See [Givón, 95] for an introduction and [Givón, 93] for a functionalist 'grammar' of English.

4 See [Haegeman, 91] for an introduction.

5 This is not to say that implementing a GB grammar is impossible, see [Wehrli, 88] for an example.

6 I will in the following sections use the terms 'word' and 'symbol' in the same sense that they are used in the literature, i.e. a symbol is an atomic unit and a word is a string of such symbols. This can be confusing in applications to natural languages as a symbol represents a word and a word in grammar theory is a sentence in a natural language.

- Grammars in which all productions are of the form $A \rightarrow w$ with $A \in V$ and $w = \varepsilon$ or $w = aB$ with $a \in T$ and $B \in V$ are called regular grammars.

Further subclasses were introduced later. The class of indexed grammars is a generalisation of CFGs and as such a subclass of CSGs. It adds to the grammar a set of indinces I that can be attached to variables and that are propagated from the left-hand to the right-hand side in productions. If a rule of the form $A \rightarrow BC$ is applied, all indices that are attached to A are also attached to B and C. If an index is stated on the right-hand side of a production, as in $A \rightarrow B_iC$, it will be added to the respective variable in addition to the indices propagated from the left-hand side. Further, if an index i is stated on the left-hand side, as in $A_i \rightarrow BC$, it will not be propagated to right-hand side; other indices of A will be propagated, though.

The class of LR(k) grammars, with k denoting a natural number, is a subclass of CFGs which basically restricts the degree of ambiguity allowed in the language. In the extreme case of LR(0) grammars this means that if $w \in L(G)$ no prefix of w can be in $L(G)$.

The word-problem is the problem deciding whether for a Grammar G and a word $w \in T^*$, $w \in L(G)$. It is interesting to note that each of the language classes corresponds to a prominent machine model, in the sense that for the class of languages which can be generated by grammars of one of the classes, there is a machine model that can solve the word-problem for exactly this class of languages. This classification therefore seems to be somewhat unrelated to the problem of determining the complexity of natural languages but I will try to determine the smallest class which completely subsumes the class of a natural languages. This goal is motivated within the engineering approach by the obvious fact that with increasing restrictions on the type of rules the word-problem can be solved more efficiently.

1.2.3 The grammar class of natural languages

The class of languages generated by regular grammars is equivalent to the class of languages accepted by deterministic finite automatons (DFA) and therefore the word-problem can be solved in time $O(|w|)$. The problem is that DFAs cannot recognize, for example, the language $L = \{ a^n b^n \mid n \geq 1 \}$ which resembles nested subordinate sentences in natural languages. It is sometimes claimed, especially by advocates of connectionist approaches to NLP, that due to performance limitations humans cannot process infinitely nested subordinate sentences either, but as natural languages theoretically allows for such constructs and we are working under the assumption of ideal speakers and listeners, this objection must be discarded.

There is no evidence that the class of languages generated by CSGs is not a superset of natural languages. In fact, the same holds true for indexed grammars.[7] Unfortunately, CSGs correspond to nondeterministic Turing machines with linearly bounded tape which implies that the word-problem for these languages is NP-complete.

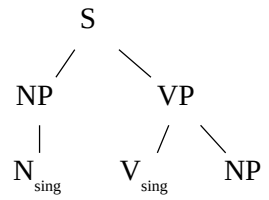
The class of CFGs provides a good compromise between tractability and expressiveness. It is equivalent to the class of languages accepted by nondeterministic push-down automatons (PDA) and there are several efficient algorithms to solve the word-problem. The most well-

7 For a discussion of potential problems see [Gazdar, 88].

known are probably the Cocke-Younger-Kasami algorithm[8] and Earley's algorithm[9] which both run in $O(|w|^3)$.

CFGs, or phrase structure grammars, were actually introduced with natural languages is mind and it is widely assumed that they are sufficient to generate natural languages. There are, however, certain phenomena in natural languages, such as agreement between subject and verb, that seem to demand for context-sensitive rules. Consider the following grammar fragment for sentences with transitive verbs and a sample parse-tree:

$S \rightarrow NP\ VP$
$NP \rightarrow N_{sing}$
$NP \rightarrow N_{plur}$
$N_{sing}\ VP \rightarrow N_{sing}\ V_{sing}\ NP$
$N_{plur}\ VP \rightarrow N_{plur}\ V_{plur}\ NP$



A look at the parse-tree reveals the problem: Context-sensitive rules are needed because there is no means to share information between the subtrees of S about which number the subject has. Consequently, a way to remove the need for the context-sensitive rules is to 'lift' the feature through the rules. The variable NP is replaced by two different variables, namely NP_{sing} and NP_{plur}, which can only generate N_{sing} and N_{plur} respectively. Following this schema one could arrive at the following context-free grammar:

$S \rightarrow S_{sing}$
$S \rightarrow S_{plur}$
$S_{sing} \rightarrow NP_{sing}\ VP_{sing}$
$S_{plur} \rightarrow NP_{plur}\ VP_{plur}$
$NP_{sing} \rightarrow N_{sing}$
$NP_{plur} \rightarrow N_{plur}$
$VP_{sing} \rightarrow V_{sing}\ NP$
$VP_{plur} \rightarrow V_{plur}\ NP$

If words bear more than one feature the number of new variables, and thus rules, is increased even further. A ruleset that would ensure agreement in person and number would make six different variables derived from NP necessary: $NP_{1st,\ sing}$, $NP_{2nd,\ sing}$, $NP_{3rd,\ sing}$, $NP_{1st,\ plur}$ etc. It is due to this property that this procedure is referred to as 'multiplying out' features.

While providing a means to transform certain CSGs into weakly equivalent CFGs, i.e. they generate exactly the same languages with a potentially different ruleset, this transformation also has some unpleasant consequences: The word-problem can be solved with a more efficient algorithm but the ruleset can become exponentially larger. In this context it is important to be aware that the running time of the algorithms is given in terms of the length of the word because it is assumed that the size of the grammar remains constant but not because the running time is independent of the size of the grammar. For example, the Cocke-Younger-Kasami algo-

8 [Younger, 67]
9 [Earley, 70]

rithm has a running time of $O(|R|\,|w|^3)$. In practice, however, it still seems to be more efficient to use the context-free version.

For the theoretical approach this procedure demonstrates that the seemingly problematic property of agreement in natural languages can be expressed with context-free grammars. In my opinion this is anything but a success as I find it very doubtful that this models the way agreement is handled in the brain. I would argue that in the light of this and further evidence[10] CFGs are not an adequate model for the theoretical approach.

CFGs are, however, extremely useful for practical purposes in NLP. This is especially true with the domain of IE in mind. For applications in this domain it is irrelevant whether the grammar is overgenerating, i.e. $\exists w \in T^*$ such that $w \notin L$ but $w \in L(G)$, as long as it can generate every word of the language, i.e. is not undergenerating, or $U = \varnothing$. This is a valid assertion because it

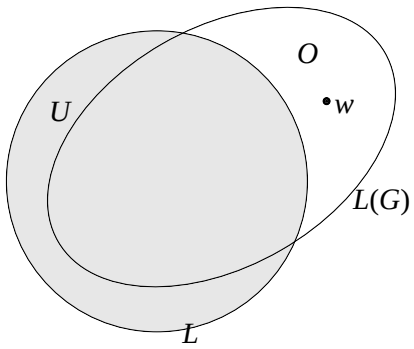


Figure 1. Over- and undergeneration.

is not the goal to determine whether a sentence is syntactically correct but to find the syntactic structure for a sentence that is assumed to be correct. As an example of the far reaching consequences of this assertion consider again the problem of agreement.

$S \rightarrow NP\ VP$

$NP \rightarrow N$

$VP \rightarrow N\ V\ NP$

This simplified grammar can generate ungrammatical sentences such as “I sees a balloons” which proves that it is overgenerating but this is not harmful as it still assigns the correct structural description to all input sentences.

It is also interesting to note that Adriaans argues[11] that the context-free version has certain properties which make it preferable for language learning tasks.

1.2.4 The lexicon component and parts-of-speech

Natural languages have a large vocabulary and linguists agree that the words can be grouped into word classes or ‘parts-of-speech’ (POS). Ideally, the members of each class are inter-

10 See [Shieber, 85] for an example of a construction in Swiss-German which provably cannot be represented with a CFG. That is, unless one takes performance limitations into account which in turn is impossible as outlined in the discussion of regular grammars.

11 In [Adriaans, 92], p. 34.

changeable without destroying the syntactic correctness or changing the structure of a sentence; the semantics will usually change, though. Consider the following example:

(1) I saw the **red** balloon.
(2) I saw the **yellow** balloon.
(3) *I saw the **and** balloon.
(4) I saw the **child's** balloon.

Replacing "red" with another (colour) adjective preserves the syntactical correctness but sentences (3) and (4) demonstrate that replacing it with a word from another class renders the sentence ungrammatical or changes the underlying structure.

Some of the classes are 'open,' i.e. new words are constantly added to them, others are considered 'closed.' An example of the former are nouns while function words such as prepositions are members of closed classes.

It is difficult to determine the exact number of classes because it is difficult to classify words on the basis of their syntactic properties as these often depend on their semantic properties.[12] In dictionaries the number of classes is less than 20 or 30 and contains prepositions, determiners, verbs and nouns with various subclasses. POS-tagging algorithms, i.e. algorithms that determine the POS for words in a text, usually work with a similar amount of classes.

These observations led to the introduction of a lexicon component L to CFGs. Instead of writing a special rule for each word, or each permutation of words, 'pre-terminals' representing the parts-of-speech are introduced. They replace the terminals in all rules and a lexicon component is used to assign terminals to the pre-terminals. As an example consider the following CFG fragment and its corresponding form with a lexicon:

(1) P = {NP $\rightarrow$ *the balloon*, NP $\rightarrow$ *a red balloon*, NP $\rightarrow$ *the red balloon*, …}
(2) P = {NP $\rightarrow$ *Det N*, NP $\rightarrow$ *Det Adj N*},
    L = {the[Det], a[Det], red[Adj], balloon[N], …}

Of course, this convention is not beyond the power of CFGs as the lexicon could be expressed with extra rules:

(3) P = {NP $\rightarrow$ D N, NP $\rightarrow$ D A N, D $\rightarrow$ *the*, D $\rightarrow$ *a*, A $\rightarrow$ *red*, N $\rightarrow$ *balloon*, …}

In an analogous way most grammar formalisms derived from CFGs can be modified to make use of a lexicon.

1.2.5 Recent grammar formalisms for NLP

As mentioned before, CFGs are a compromise between tractability and expressiveness. It is thus only natural that for certain applications new restrictions on the grammar rules were introduced in favour of making parsing more efficient. Designers of programming languages usually restrict themselves to the class of languages generated by LR(1) grammars which are equivalent to the class of languages accepted by deterministic push-down automatons. Algorithms for

12 The Emile algorithm can handle arbitrary numbers of classes. There are, however, certain considerations that have to be taken into account. See paragraph 3.2.3.

the word-problem, shift/reduce-parsers for example, work in O($|w|$). While this class is too small for natural languages as phenomena like PP-attachment-ambiguity cannot be represented, it motivated Tomita's parsing algorithm[13] which has the following properties:

- It cannot parse words in grammars that have cyclic rule subsets, i.e. $S \to S$ and $S \to x$ or are infinitely ambiguous, i.e. include rule subsets of the form $S \to S\,S$, $S \to \varepsilon$ and $S \to x$.
- If the grammar is heavily ambiguous its running time can be greater than O($|w|^3$) but if the grammar is a LR(1) grammar it has a running time of O($|w|$).

Masaru Tomita argues that cyclic rules and infinite ambiguity do not occur in natural languages, that heavy ambiguity is rare and that natural languages are 'close' to languages generated by LR(1) grammars. Hence, his parsing algorithm should be efficient while still being able to cope with the full expressiveness of natural languages. For exactly this reason Tomita's algorithm has gained great popularity in practical applications and I will return to it later.

CFGs have also been extended in certain ways to increase expressiveness. A well-known development in this area are definite clause grammars (DCGs)[14] which are embedded in Prolog. DCGs have a context-free backbone in which rules can be annotated with arbitrary terms. The unification mechanism of Prolog provides an elegant means to handle phenomena such as agreement and also allows for the construction of alternative representations while parsing a sentence. This comes at a high price though: the expressiveness of DCGs is equivalent to the expressiveness of Prolog which means that the word problem corresponds to the halting problem for Turing machines. In practical applications the following parsing algorithm, also known as 'generate and test,' can be employed. First, a candidate structure is generated using the CFG rules, then the attached terms are evaluated. While this is a feasible approach, it is far from being efficient in comparison to parsing algorithms for pure CFGs. PATR-II[15] and Gulp[16] are other popular grammar formalisms with a similar background.

1.3 Stochastic language processing

1.3.1 Supplements to context-free grammars

After an enormous success in speech recognition the use of stochastic techniques for all levels of language processing is being explored.[17] While there are language models that are completely statistical, such as Hidden Markov Models or N-grams, there is one approach that is of particular interest to us: Stochastic or probabilistic context-free grammars (SCFGs) simply associate a probability $P(A \to B)$ with each rule stating the probability of expanding the variable A using this rule as opposed to any of the other rules with A on the left-hand side. Consequently, the probabilities for all rules with the same variable on the left-hand side must sum to one.

13 [Tomita and Ng, 91] and [Tomita, 86]

14 [Pereira and Warren, 80]

15 [Shieber *et al.*, 83]

16 [Covington, 91]

17 For an overview see [Charniak, 95].

The probability $P(w, t)$ of a parse-tree t for a word w is the product of the probabilities of the rules used in its generation. Note that if a rule is used more than once, its probability is included for each application. The overall probability for a word $P(w)$ is the sum of the probability of all of its parses.[18]

As SCFGs add nothing to the expressiveness of CFGs, parsing algorithms suitable for CFGs are also suitable for SCFGs. All that is needed are additions to keep track of the probabilities when applying the rules.[19]

1.3.2 Comparison with the standard approach

An obvious advantage of SCFGs over CFGs could be expected in cases of syntactic ambiguity. A parser for the SCFG does not only generate all parse-trees but also establishes a ranking of them and it could be assumed that the parse-tree with the highest probability corresponds to the 'correct' or most intuitive parse. Unfortunately, this is not the case as SCFGs simply favour common constructions over less common ones which while being an important factor is obviously not the only one. This problem can be illustrated with the following example of two PP-attachment ambiguities:

(1) Alice bought a plant with Mary.
(2) Alice bought a plant with yellow leaves.

Statistically, constructions like the one in example (2) occur more often but this would provide no guidance for a parser because in this case the key factor is the NP under the PP and specifically the knowledge whether it can be a part of a plant or not.

In practical applications SCFGs do have an advantage when dealing with large corpora. It is likely that any large enough corpus will contain ungrammatical sentences which are, despite their syntactic defects, understandable. A CFG-based algorithm will simply reject such sentences and by that miss the information in them. An SCFG on the other hand can be extended by a couple of rules that can generate an arbitrary string of symbols with a very low probability. These rules can then by applied to cover the ungrammatical patches.

Most importantly, though, it is a supposition that SCFGs could provide a means to solve one of the main problems of grammar learning.This will be discussed in paragraph 2.2.4.

1.4 Categorial grammars and the Lambek calculus

1.4.1 Introduction

The term 'categorial grammar' (CG) is a collective term which covers a number of related formalisms which are all generalisations of a core grammar formalism which was first explicitly defined by Ajdukiewicz[20] but has its origin in the works of Husserl, Lesnevski, Frege, Carnap

18 A proof for these claims can be found in [Charniak, 95], pp. 75–79.

19 See [Fujisaki *et al.*, 89] for a modified Cocke-Younger-Kasami-algorithm and [Briscoe and Carrol, 93] for an adaptation of Tomita's algorithm.

and Tarski on semantic and syntactic categories. These formalisms have a number of distinguishing features that clearly sets them apart from the Chomskian approach. They have been proposed for syntax as well as semantics with the assumption of a close relationship between syntactic and semantic types, the former often merely encoding the latter. They also do not employ a set of rules for the syntax of the language but encode it in the lexicon and the constituents are characterised syntactically and semantically as functions and/or arguments.

The theoretical assumptions underlying categorial grammars can be found in much modern work on theories of natural language semantics. In comparison with other theories of syntax, it was claimed that categorial grammars have significant advantages as explanatory and unifying theories of natural language phenomena such as unbounded constructions including coordination and relative clause formation. Therefore they have, in the early nineties, often been considered a source of alternatives to transformational rules and their derivatives for modern formal theories of natural language syntax. It is also interesting to note that the idea to represent syntax in the lexicon, which is contradictory to the concept of CFGs, has to some extent been mimicked by most recent NLP systems. These systems provide type and subtype information for word types, most notably for verbs, in the lexicon and this information is used somewhere in the ruleset to trigger certain special behaviour. In fact, some implementations seem to have at least as many rules for VPs as they have verbs in the lexicon.

1.4.2 Pure categorial grammars

In a categorial grammar, all grammatical constituents, and in particular all lexical items, are associated with a 'type' or 'category' which defines their potential for combination with other constituents to form compound constituents. This type can either be from a small set of basic categories, such as NP, or it can be a 'functor' type. The latter defining a function that maps arguments of some type onto results of some possibly different type. For example, the type of an English intransitive verb is most naturally defined as a function from a NP on the left of the verb to a sentence. A transitive verb can then be defined as a function from a NP on the right of the verb to the type for intransitive verbs. As mentioned before, CGs represent the syntax in the lexicon and thus need no equivalent to the production rules of CFGs. They do, however, have two invariant rules for functional application:[21]

(1) $x/y \;\; y \;\; \Rightarrow \; x$
(2) $y \;\; y\backslash x \Rightarrow \; x$

A constituent type like x/y states that this constituent together with a constituent of type y on its right yields a constituent of type x. Rule (1) is the definition of this. Similarly, rule (2) defines the converse case: A constituent of type $y\backslash x$ can be combined with a constituent of type y on its left to yield a constituent of type x. Note that in this notation directionality is encoded in the slash as well as in the position of the argument: A forward slash indicates a rightward argument which is stated on the right of the slash and a backslash indicates a leftward argument given on the left of it.

The following example presents CFG rules for some simple sentences with transitive and intransitive verbs. (The problem of agreement is ignored for the moment.)

20 [Ajdukiewicz, 35]. A translation can be found in [McCall, 67].

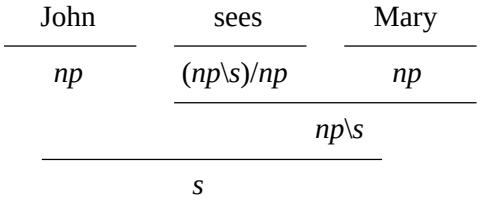
21 There are several different notations. I will use a variation of Lambek's 'slash notation.'

S → NP VP
VP → VI, VP → VT NP
NP → *John*, NP → *Mary*
VI → *walks*
VT → *sees*

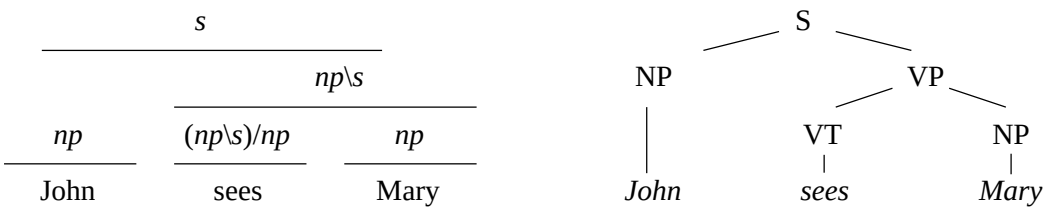
The same language can be described by the following lexicon in a CG:

⟨John, *np*⟩
⟨Mary, *np*⟩
⟨walks, *np**s*⟩
⟨sees, (*np**s*) / *np*⟩

With this lexicon the analysis of the sentence “John sees Mary” can be understood as a successive application of the functions in the functor types as illustrated in the following figure.



An interesting question is that about the generative power of CGs. A comparison of the parse tree of the sentence and the functional applications (written in ‘generating’ order) reveals a striking similarity and suggests that CFGs and CGs are weakly equivalent.



It was believed since the early sixties when Bar-Hillel and Lambek were developing the first CGs that this equivalence held but it was not until 1993 that this conjecture was proved by Mati Pentus.[22]

In the discussion of the standard approach I mentioned that CFGs are not sufficient to model all phenomena found in natural languages. Of course the same holds true for CGs and therefore a variety of generalisations were developed some of which incorporated concepts such as transformational rules from the Chomskian approach. Others introduced further grammar rules to deal with these phenomena in an ad hoc manner. Well-known examples are rules for raising and cancelling with which complex coordination as in “John gave Mary a book and Susan a record.” can be handled.

(1) Raising: $(y/x) \setminus y \Rightarrow x$ and $y / (x \setminus y) \Rightarrow x$
(2) Cancelling: $x/y\ y/z \Rightarrow x/z$ and $z \setminus y\ y \setminus x \Rightarrow z \setminus x$

22 [Pentus, 93]

Other approaches to extend CGs are motivated by a mathematical view that I will describe in the next section.[23]

1.4.3 A mathematical language model

Historically, CGs are the result of an attempt to develop a syntactic calculus. Starting with the observations that sentences are constructed from words by concatenation and that the sequence of the words in the sentence is relevant, Lambek introduced a non-commutative concatenation operator and proposed three structural levels to study a language:

- A multiplicative system. Defined by the set of basic types and the concatenation operator. Some of the concatenations are grammatical, others are not. The operator is binary and non-associative which implies that sentences must be bracketed. This corresponds to a tree-like structure.
- A semigroup. The multiplicative systems with the operator being associative. Therefore, sentences are not bracketed and have less structure.
- A monoid. The semigroup with a unity element, the empty word.

The subsets of a multiplicative system M are subject to three operations: The dot is the concatenation operator and the two slash operations are defined to behave like they do in a CG.

$$A \bullet B = \{\, xy \in M \mid x \in A \text{ and } y \in B \,\} \tag{1.1}$$

$$\mathrm{C} / \mathrm{B} = \{\, x \in M \mid \forall y \in B\colon xy \in C \,\} \tag{1.2}$$

$$\mathrm{A} \setminus \mathrm{C} = \{\, y \in M \mid \forall x \in A\colon xy \in C \,\} \tag{1.3}$$

The slash operators add no further structure as they can be expressed in terms of the concatenation operator. This fact can be illustrated by the following equivalencies which hold true for all A, B and C in M:

$$A \bullet B \subseteq C \text{ iff } A \subseteq C / B \tag{1.4}$$

$$A \bullet B \subseteq C \text{ iff } B \subseteq A \setminus C \tag{1.5}$$

For a semigroup M the associativity of the operator can be expressed by equivalency (1.6) and for a Monoid M with the unity element 1 and $I = \{1\}$ equivalency (1.7) holds true.

$$(A \bullet B) \bullet C = A \bullet (B \bullet C) \tag{1.6}$$

$$I \bullet A = A = A \bullet I \tag{1.7}$$

The notation $f : A \to B$ denotes a proof f for the fact that A is included in B. There are two structural rules for proofs: The identity arrow I_A for each type A and the composition of arrows.

$$I_A\colon A \to A \tag{1.8}$$

$$\frac{f\colon A \to B \quad g\colon B \to C}{gf\colon A \to C} \tag{1.9}$$

On this basis the Standard Lambek calculus can be defined by the following axioms and rules:

23 For collections see [Buszkowski *et al.*, 88] and [Oehrle *et al.*, 88]. An overview of the whole area is presented in [McGee Wood, 93].

$$\alpha_{A,B,C}: (A \bullet B) \bullet C \rightarrow A \bullet (B \bullet C) \quad \text{and} \quad \alpha^{-1}_{A,B,C}: A \bullet (B \bullet C) \rightarrow (A \bullet B) \bullet C \tag{1.10}$$

$$\rho_A: A \bullet I \rightarrow A \quad \text{and} \quad \rho^{-1}_A: A \rightarrow A \bullet I \tag{1.11}$$

$$\lambda_A: I \bullet A \rightarrow A \quad \text{and} \quad \lambda^{-1}_A: A \rightarrow I \bullet A \tag{1.12}$$

$$\frac{f: A \bullet B \rightarrow C}{f^*: A \rightarrow C/B} \quad \text{and} \quad \frac{f: A \bullet B \rightarrow C}{{}^*f: B \rightarrow A\backslash C} \tag{1.13}$$

$$\frac{g: A \rightarrow C/B}{g^*: A \bullet B \rightarrow C} \quad \text{and} \quad \frac{g: B \rightarrow A\backslash C}{{}^*g: A \bullet B \rightarrow C} \tag{1.14}$$

The axioms in (1.10) take care of the associativity. Axioms (1.11) and (1.12) represent the identities stated in (1.7). The arrow transformation schemes in (1.13) and (1.14) corresponds to the equivalencies in (1.3) and (1.4). Note that in a calculus like this the following always holds true: $(f^*)^+ = f$ and $^*(^+g) = g$.

The Lambek calculus is well suited for linguistic purposes as it lacks certain rules usually found in other calculi. For example, the Lambek calculus has no interchange rule which would allow interchanging of words within a sentence. But as this usually leads to ungrammaticality, the absence of this rule in the Lambek calculus is a sensible choice. Another important property of the Lambek calculus is that Lambek could prove its decidability for all levels, i.e. with and without an associative concatenation operator and unity element. Finally, the Lambek calculus is structurally complete[24] which means that the following condition holds: If a sequence of categories $X1, \ldots Xn$ reduces to Y there is a reduction to Y for any bracketing of $X1, \ldots Xn$.

The value of the Lambek calculus lies in the fact that it allows the study of parsing as a form of deduction. In this case it is important to note that a type is nothing but a set of expressions. Returning to the example from the section on pure CG the following axioms might be given.

g : {John} $\rightarrow N$
f : (N • {walks}) $\rightarrow S$

From the second axiom the type of “walks” can be derived to be

$^*f$: {walks} $\rightarrow N\backslash S$

Now the sentence “John walks” can be derived as follows, showing how the calculus acts as a grammar:

$$\dfrac{g: \{\text{John}\} \rightarrow N \qquad \dfrac{\dfrac{\dfrac{{}^*f: \{\text{walks}\} \rightarrow N\backslash S}{{}^+({}^*f): N \bullet \{\text{walks}\} \rightarrow S}}{f: N \bullet \{\text{walks}\} \rightarrow S}}{f^*: N \rightarrow S/\{\text{walks}\}}}{g(f^*): \{\text{John}\} \rightarrow S/\{\text{walks}\}}$$

$$\dfrac{g(f^*): \{\text{John}\} \rightarrow S/\{\text{walks}\}}{(g(f^*))^+: \{\text{John}\} \bullet \{\text{walks}\} \rightarrow S}$$

24 [Buszkowski, 88]

After this formal definition of the Lambek calculus I can present Lambek's definition of a CG. According to him a categorial grammar for a language consists of a syntactic calculus and a dictionary. The calculus is 'freely generated' from a finite set of basic types $BT = \{ T_1, \ldots T_n \}$, i.e. the set of types T is defined inductively from $BT \cup \{ I \}$ by the operations "•", "/" and "\" and the set of proofs is freely generated from the axioms by the rules of inference of the Lambek calculus. The dictionary associates each word w_i with a set of types $WT_i \subseteq T$. With this understanding of a grammar Lambek defines the type of a string of words $w_1 w_2 \ldots w_n$ to be S if and only if $WT_1 WT_2 \ldots WT_n \rightarrow S$ is a theorem freely generated by the syntactic calculus. He considers a grammar to be adequate provided it assigns the type S to a string of words $w_1 w_2 \ldots w_n$ if and only if the latter is a grammatical sentence.

1.4.4 Extensions to the Lambek calculus

For a formalism to be suitable for a learning task it must able to deal well with partial information. The standard Lambek calculus already allows for the deduction of partial information about set inclusions but this is not sufficient for the Emile learning algorithm. I will introduce two new operators proposed by Adriaans for this purpose.[25] The join or supremum operator and the negation or complement operator are defined as followed:

$$A \vee B = \{ x \in M \mid x \in A \text{ or } x \in B \} \tag{1.15}$$

$$\neg A = \{ x \in M \mid x \notin A \} \tag{1.16}$$

Corresponding to these new operations there are new axioms and rules in the calculus:

$$\kappa_{A,B} : A \rightarrow A \vee B \quad \text{and} \quad \kappa'_{A,B} : B \rightarrow A \vee B \tag{1.17}$$

$$\frac{f : A \rightarrow C \quad g : B \rightarrow C}{[f, g] : A \vee B \rightarrow C} \tag{1.18}$$

$$\neg\neg A \rightarrow A \tag{1.19}$$

As he is also interested in semantic learning, Adriaans introduces more operands to define an Extended Lambek calculus. He does prove the decidability of the calculus but states that this is of little practical value as the corresponding model is extremely large.

25 For an explanation for the choice of these operators and further operators that deal with semantic learning see [Adriaans, 92], pp. 61–78.

Chapter

2 Language learning

This chapter serves as an overview of the field of language learning and an explanation of the approach presented in this thesis. After a brief treatment of some relevant concepts from the 'mother discipline' machine learning (ML) in section 2.1 I will present other approaches to the task of language learning in general and to grammar induction in particular. An evaluation of these will lead to Adriaans' approach to language learning which is the subject of section 2.3. In this section I will present the necessary theory and provide a formal definition of cooperative language learning.

2.1 Background

2.1.1 Machine learning

The ability to learn is probably the most prominent display of intelligence.[1] Furthermore, shortly after the availability of the first computers it was believed that learning would play a dominant role not only in artificial intelligence but in all fields of computer science because it seemed impossible to explicitly program every detail of the software systems that were envisioned.[2] Obviously, no distinction was yet made between program and data in terms of the von Neumann model of computers and it took until the mid seventies with the rise of expert systems that a separation into a knowledge base and an interpreter was introduced. The knowledge base consists of an open set of rules in a formalism predetermined by the interpreter, production rules or first order predicate calculus (FOPC) for example. The interpreter or inference engine is a system that can derive conclusions from the knowledge base. As the interpreters and their properties depend on the chosen formalism, which is usually well understood, they leave little room for qualitative improvement and, consequently, machine learning has concentrated on the creation and especially the improvement of the knowledge bases. And it is this understanding of machine learning that is employed throughout this thesis as it is the goal to learn a grammar, i.e. a set of rules, and not a system that utilizes them. Also, the formalism will be determined in advance.

2.1.2 A definition of learning

Research in ML has also raised some fundamental questions about learning. Even the very definition about what it means to 'learn' something is subject of an ongoing debate. While it seems generally accepted that learning is a process that creates or modifies representations of some kind it remains uncertain whether a direction in this process is required. The problem is that it is part of the intuitive understanding of learning that it entails improvement but it is difficult to

1 And, in fact, the charter of the at Dartmouth conference lists learning as the prime example for intelligence.

2 [Turing, 50], reprinted in [Boden, 90].

asses the result of a learning process without making an assumption about how it will be put to use. Or in other words, it is difficult to say whether something was learnt, unless it is applied.[3]

The subject of the learning process in this thesis are grammars and it is one of their properties that they define a language. This allows us to avoid the dilemma of requiring an application as we can always make a hypothetical application by comparing the sets of sentences that our grammar can generate and those which are considered grammatically correct in our target language. Therefore, I will consider learning to have taken place if a grammar covers more sentences that are considered grammatically correct and/or fewer sentences that are considered incorrect. As outlined at the end of paragraph 1.2.3 the latter criterion can be dropped for certain applications.

2.1.3 Forms of learning

There are several forms of learning, ranging from 'learning by being told' to 'learning by discovery.' The former means that the student, or the entity which is learning, is told explicitly was is to be learned. This implies that teachers must transfer their knowledge into a representation which is directly acceptable for the student. An extreme example of this kind of learning might be programming in the general sense, as the teacher (programmer) tells the computer (student) explicitly what to do. The main burden lies on the teacher and this is what makes this option less attractive. At the other extreme, in learning by discovery, no teacher of any kind is involved and the student has to acquire concepts from unstructured observations in the world, or a virtual world. In this case the burden is obviously on the learner. While this type of learning seems desirable because it requires no activity other than modelling the world, it suffers from the problem that it often cannot be used to solve specific problems as the representation found by the learner can be opaque and hard to relate to the original problem.

It should be obvious now that some intermediate form of learning is needed; one that distributes the burden between teacher and student and one that leaves the option for the teacher to trade expenditure against specificity. And, in fact, a form of learning that allows for this is the most researched form of machine learning. It is usually known as inductive learning or, along the lines of the previous paragraph, it could be called 'learning by example.' The teacher performs the aggregation step, i.e. classifies the observations and presents them as positive and negative examples to the student. It is then up to the student to process the examples and find a consistent hypothesis. Or, more formally, given a language L_e to describe the examples, a language L_H in which hypotheses can be expressed, a set P of positive examples, a set N of negative examples and a predicate *covers* that classifies the examples, the student has to find a hypothesis $h \in L_H$ such that $\forall p \in P$, *covers*(h, p) and not $\exists n \in N$, *covers*(h, n). The Emile algorithm searches for a hypothesis, notably a most general hypothesis, that is consistent with strings of words which are classified as either syntactically correct or incorrect.

2.1.4 The problem of learning a language

At first sight learning a language seems to be a task which cannot be decomposed easily. When children acquire their first language all that is observable is that they are exposed to utterances in a language and after a while they are able to speak that language. Hence, they must have

3 For a discussion see [Morik, 95], p. 244.

learned it somehow. The process becomes less opaque when one examines the way adults learn foreign languages. It is obvious that they can talk about language, i.e. they have ways to represent knowledge about language. With grammars for example they possess a formalism to describe the structure of sentences and by that can state rules that govern the construction of them. Thus, one way of learning a language could be to ask a teacher for the rules of a language and add them to the ones acquired previously. While it is arguable that this is the way languages are learnt it shows at least that to be able to learn a language two prerequisites are required: A means to represent knowledge about language and a procedure to build or extend representations of knowledge from input in some form. With grammar formalisms we have the former and with learning by example we have the latter. What is missing is a combination of the two.

2.2 Grammar induction

2.2.1 Distributional analysis

An obvious choice for a grammar formalism are CFGs and positive examples are readily available in the form of texts. Providing a sufficient amount of negative examples seems less obvious. With this background Lamb developed in the sixties one of the first algorithms[4] with the goal to learn a phrase structure grammar solely by analysing texts in the target language. It was an application of the 'distributional analysis' of Harris and Hockett[5] and as such makes the following assumption: Phrase structures that are found in the same contexts define a phrase category. Contexts that share the same categories are considered to be equivalent and complex phrase structure categories are constructed by the concatenation of simple categories. A similar assumption is made in the Emile algorithm.

2.2.2 Identification in the limit

In disagreement with the Chomskian approach Mark Gold believed very early that research in NLP should be done on the basis of performance data rather than the rules that constitute competence:

> "Since we cannot explicitly write down the rules of English which we require one to know before we can say he can 'speak English,' an artificial intelligence which is designed to speak English will have to learn its rules from implicit information. That is, its information will consist of examples of the use of English and/or of an informant who can state whether a given usage satisfies certain rules of English, but cannot state these rules explicitly."[6]

I still consider this a very reasonable argumentation and also there is a recent revival of approaches to language learning under this assumption, especially in the field of stochastic language processing. Therefore, the results of Gold's work should be of great interest.

4 [Lamb, 61] as cited in [Gold, 67]

5 [Harris, 51], [Harris, 64] and [Hockett, 58]

6 [Gold, 67]

In order to make the language learning problem mathematically tractable, Gold introduced the concept of identification in the limit. First a set of languages is specified. Each of which is taken to be a set of strings, or words in our terminology, over the same finite alphabet. Also, there is a naming relation to unambiguously identify each language. A teacher chooses a language and a method of presentation. Gold proposes two methods with three subtypes but for our purposes it is sufficient to distinguish between presentations that include negative examples (informant) and those that are made up of positive ones only (text). The learning session starts at a time t_0 and continues forever. At each time t_n the learner receives a unit of information from the teacher and makes a guess about the name of the language on the basis of all the information received so far. A class of languages is considered learnable with respect to the given method of presentation if there is an algorithm available to the learner which has the following property: Given any language of the class there is some finite time t_l after which the guesses regarding the name of the language are all the same and are all correct. Note that this implies that it is not necessary for the learner to know that he has identified the language.Within this learning model Gold proved the results summarized in figure 2.

Method of presentation	Class of languages
Informant	Recursively enumerable Recursive Primitive recursive Context-sensitive
Text	Context-free Regular Finite cardinality

Figure 2. Gold's learnability results

The language classes are listed in an order such that each class is properly contained in the class above. It should be obvious that if a class of languages is identifiable in the limit the same holds true for any subclass and that if a class is not identifiable in the limit the same holds true for any superclass. Therefore it is possible to draw dividing lines with the meaning that all languages below it can be identified in the limit with respect to the method of presentation.

Undoubtedly, the most important result for the grammar learning community was that the classes of languages defined by CFGs and even regular languages cannot be learnt from positive example alone. Gold's proof is basically by contradiction: Any proposed guessing algorithm has to identify any finite language after a finite amount of text. Each of the language classes from regular languages up contain languages with an infinite cardinality. A text for an infinite language can be constructed in such a way that it ranges over successively larger, finite subsets of the language. At each stage it repeats the elements of the current subset long enough to fool the learner, but then advances to the next larger subset. The algorithm therefore has to exploring an infinite number of false hypotheses and can never identify the language.

Gold's result had a paralysing effect on research in this area, comparable to the effect that Minsky's critique of the Perceptron had on connectionist computing. It should be noted that while being interesting from a theoretical point of view, the practical value of Gold's findings is limited. For example, consider two other theorems proved in the same paper: The first one states

that no learning algorithm is uniformly better than any other and the second one states that the teacher's behaviour, i.e. the sequence of the words during the presentation, is irrelevant. In practical applications, however, an evaluation of the teacher's behaviour is of great importance. Of course it is also unrealistic to assume infinite learning time. In essence, Gold's results simply provide extreme borderlines but what is needed is a framework to evaluate various practical learning strategies, including an analysis of the teacher's behaviour and teacher-learner interaction.

2.2.3 Grammar induction after Gold

As Gold had proved that none of the interesting language classes could be learnt from positive examples only, researchers investigated with which, presumably natural constraints learnability could be achieved. Abe and Yokomori present such constraints for CFGs.[7]

Most other research considered extended student-teacher interaction. It is obviously desirable to have algorithms that can learn from (positive) examples but it is just as obvious that in absence of such algorithms learning does not become impossible or uninteresting.

Dana Angluin showed[8] that regular languages can be learned if the algorithm can make equivalence and membership queries. She also proved that the learning time is polynomial in the number of states of the minimum deterministic finite automaton for the language and the maximum length of any counter-example. The question whether an analogous result for CFGs can be found is still open. With the inclusion of non-terminal membership queries Angluin could provide a partial solution.[9] Building on Angluin's work, Yasubumi Sakakibara developed an algorithm[10] that learns CFGs in polynomial time. However, he makes use of structural equivalence queries and structural membership queries which effectively means that he needs structural data, i.e. unlabelled derivation trees, instead of unstructured examples.

Robert Simmons and Yeong-Ho Yu present another approach that works with structured input and an interaction with a teacher.[11] Basically, their system predicts a parse for each example sentences based on the input received so far and expects the teacher to correct it. This is probably the strongest form of cooperation from the teacher and it seems on the borderline between learning by example and learning by being told. The results of their work, however, are promising, even when taking the immense burden on the teacher into account. The core of their system is a parser motivated by shift/reduce parsers. It contains a heuristic component and can parse a class of languages that the authors call context-dependent grammars, an actual superset of CFGs. Their system scales well and can process rather complex sentences.

2.2.4 Identification by enumeration

Another conceivable strategy for grammar induction is 'identification by enumeration:' The language is guessed to be the simplest one that accounts for all examples presented. Gold deals

7 [Abe, 88] and [Yokomori, 89]

8 [Angluin, 87]

9 [Angluin, 87b]

10 [Sakakibara, 88]

11 [Simmons and Yu, 92]

with this strategy as well and demonstrates that the problem lies in the search for the 'simplest' language. He provides a counterexample in which the algorithm is forced to make only wrong guesses. The proof is again a borderline case but nevertheless it is instructive as it is based on a fact that causes problems in practical applications: A superset *L'* of the language *L* is consistent with any finite segment of any text for *L*. Or in other words, an algorithm can guess a too large language and never receive an indication of that.

In a practical application an algorithm might guess that the following grammar is the grammar of English and never discover any evidence against it.

$S \rightarrow W\ S$

$S \rightarrow W$

$W \rightarrow$ the, $W \rightarrow$ cat, $W \rightarrow$ sat, …

While this is an extreme example and it will be feasible to find a simpler grammar, it remains unclear how the simplest grammar can be found. SCFGs might provide a solution to this problem as they not only assign a parse but also a probability to each sentence. An algorithm could make several hypotheses about the grammar of a language and then rank them according to the probabilities assigned to the entire text. This approach seems reasonable because a grammar that is similar to the example one does, as it is overgenerating, assign non-zero probabilities to a number of sentences not contained in the text and therefore will also assign lower probabilities to the examples in the text. Therefore, the simplest language could be defined as the language generated by the grammar that assigns the highest probability to the text. The key idea of this approach is to utilise the a priori probability of hypotheses, an idea that is generally known under the name of 'minimum description length' theory.[12]

Another approach in this area is that of James Cussens and others. They present an experiment[13] with a system that has the following properties: The formalism to represent the knowledge about the grammar, the examples and hypotheses is that of stochastic logic programs (SLP), an extension to FOPC analogous to the extensions in SCFGs. Given this powerful formalism standard techniques from the area of inductive logic programming (ILP) can be employed. The learning process starts with a small hand-coded grammar. For each sentence in a batch of examples that cannot be parsed with the background knowledge acquired so far, the system generates up to six new rules (productions) such that the example can be covered and the rules maximally compress a set of entailed examples. An oracle is then responsible for selecting which rules are to be included into the background knowledge. The oracle might also reject all proposed rules forcing the system to add a rule that marks the sentence as a special case. The authors used this system to incrementally learn a grammar from a series of children's early reader books with each book introducing more complex grammatical constructions.

2.2.5 Other approaches to ML

All approaches to language learning I have described so far can be ascribed to the 'symbolic' paradigm in ML. The distinguishing features of these approaches are a symbolic representation of knowledge and a centralised processing of it. A general paradigm-shift towards decentralised

12 [Rissanen, 78]

13 [Cussen *et al.*, 97]

systems[14] influences a growing part of the ML community and research is also conducted within various 'non-symbolic' disciplines. The systems usually employ subsymbolic representations and draw heavily on small and simple autonomous units that function in a decentralised manner. There are approaches to the problem of grammar learning in the areas of genetic algorithms[15] and especially connectionism,[16] but it is beyond the scope of this thesis to describe them. It must therefore suffice to state that the results achieved so far provide new insights for (psycho-) linguistic theory as the systems have to deal with representational issues in a novel way but are still somewhat less applicable to real-life problems because they usually do not scale too well.

2.3 Cooperative language learning

2.3.1 Introduction

The evaluation of the approaches to grammar induction in the previous section indicate that a successful attempt to learn a grammar will have to incorporate a non-trivial form of interaction between the teacher and the learner. It seems that we not only need negative examples but that it is necessary to have access to certain specific negative examples to validate hypotheses that are formed during the learning process. In terms of the student-teacher metaphor we are clearly postulating cooperation between the involved parties or, basically, a 'fair' teacher. This is not surprising as Gold's results prove that a malicious teacher can make learning of even the simplest languages impossible.

To consider language learning as a cooperative task between two parties can also be justified from observations of language use as the following example illustrates:[17]

> One day when Christopher Robin and Winnie-the-Pooh and Piglet were all talking together, Christopher Robin finished the mouthful he was eating and said carelessly; 'I saw a Heffalump today, Piglet.'
>
> 'What was it doing?' asked Piglet.
>
> 'Just lumping along,' said Christopher Robin. 'I don't think it saw me.'
>
> 'I saw one once,' said Piglet. […]

Assuming that at least Piglet does not know what a 'heffalump' is we can consider this dialogue as an example of how Piglet begins to learn a new word. From the first statement of Christopher Robin, most likely from the way the word is used in the sentence he infers that a heffalump is a physical object and he also assumes that a heffalump is some sort of agent. He could verify this hypothesis by openly asking Christopher Robin but he pursues a different strategy: He asks a different question that presupposes his hypothesis and takes the fact that Christopher Robin

14 See for example [Minsky, 86].

15 See for example [Lankhorst, 94].

16 For an overview see [Touretzky, 91], a GB inspired approach is [Berg, 92].

17 From 'Winnie-the-Pooh' as quoted in [Adriaans, 92].

accepts the question as confirmation. Christopher Robin in turn is over cooperative as he also states that heffalumps can see, another hint that they are agents.

This example exhibits a clear learning pattern: The teacher states an example. The learner establishes a hypothesis that is consistent with the example and verifies the hypothesis with the help of the teacher. In fact, it was observations like this which motivated Adriaans research and he starts his investigation into language learning with the following theory:

> "The paradigmatic situation is that of dialogue. Semantic categories are substitution classes of terms. Terms in the same semantic category can be substituted for each other salva significatione. We can analyze the structural resemblances between sentences uttered by the participants in a dialogue. Based on this analysis we can form a hypothesis concerning the possibility of substituting terms. This hypothesis can be tested by observing the reactions of other participants in the dialogue, either by uttering a statement corresponding with the hypothesis or by simply asking. Based on these reactions we can adjust our hypothesis. We have learned the language if we can predict the reaction of the participants to our questions with a reasonable degree of certainty."[18]

Note that Adriaans, in contrast to Gold, does not require an exact identification of the language as he is content with a 'reasonable' approximation. The remainder of this section provides a formal framework for this theory

2.3.2 Pac-learning

In 1984 Valiant introduced a learning model that allows for the formal evaluation of the teacher-learner interaction.[19] He originally set out to prove that it is possible to construct an algorithm, or learning machine in his terminology, that could provably learn classes of concepts which are appropriate for general-purpose knowledge and could do so in polynomially bounded time. As an example he chose to learn certain classes of Boolean functions of a set of propositional variables. This satisfies his appropriateness condition because propositional calculus was and is still used for knowledge representation in expert systems.

Valiant considers n Boolean variables $p_1, \ldots p_n$. He defines a vector v to be an assignment of a value from {0, 1} to each of the variables. A Boolean function is defined as a a mapping from the set of the 2^n vectors to {0, 1}. Given a Boolean function f he introduces two arbitrary probability distributions P and N over the set of all vectors such that

$$P(v) > 0 \text{ iff } f(v) = 1,\ P(v) = 0 \text{ iff } f(v) = 0 \text{ and } \sum P(v) = 1.$$
$$N(v) = 0 \text{ iff } f(v) = 1,\ N(v) > 0 \text{ iff } f(v) = 0 \text{ and } \sum N(v) = 1.$$

The learner can employ any algorithm and can make use of two functions:

- EXAMPLE. This function has no input and it returns a vector v such that $f(v) = 1$. For each such v the probability that v is returned on any single call is $P(v)$. Note that this routine by definition provides positive examples only.
- ORACLE(). On vector v as input it returns 0 or 1 according to whether $f(v) = 0$ or 1.

18 [Adriaans, 92], p. 8

19 [Valiant, 84]

A class of functions X is defined to be learnable if there is a learning algorithm with the following properties:

- The algorithm runs in a time polynomially bounded by an adjustable parameter h, various parameters that quantify the size of the function to be learned and the number of variables.
- For all functions $f \in X$ and all distributions P and N the algorithm deduces with probability at least $(1 - h^{-1})$ a function $g \in X$ that fulfils the following requirements:
  - If $g(v) = 1$ but $f(v) = 0$ then for all such v the following holds $\sum N(v) \leq h^{-1}$.
  - If $g(v) = 0$ but $f(v) = 1$ then for all such v the following holds $\sum P(v) \leq h^{-1}$.

It is because of the second property that Valiant's learning model has become know under the term 'pac-learning' which is simply an abbreviation for 'probably approximately correct' learning.

Valiant was able to prove, with a rather abstract lemma from stochastics, that certain classes of Boolean functions are learnable according to this definition. One of the proofs in the original paper is for the class k-CNF, the class of expressions formed by a conjunction of clauses, each of which being a disjunction of up to k literals. However, Valiant also cites evidence from cryptography that not the whole class of functions computable by polynomial size circuits is learnable and Li and Vitányi have found a number of other negative results for even trivial problems.[20]

In essence, pac-learning provides a framework to evaluate learning algorithms that is better suited to practical applications than Gold's identification in the limit. It allows for an evaluation of the teacher-learner interaction and the complexity of the learning task and also captures certain natural circumstances in practical learning situations, notably the fact that absolute certainty is not required and the behaviour on unnatural input is irrelevant. The negative results for many problems are due to one degree of freedom in Valiant's definition that can be restricted for our purposes. Instead of allowing an arbitrary distribution of positive examples we will require a distribution that is 'fair' according to some definition.

2.3.3 Kolmogorov complexity

When we are asked to describe the two strings "33333333" and "97513720", we will most likely describe the first one as 'a sequence of eight threes' while for the second one we will have to state each digit separately. Obviously, the first string has a smaller information content than the second one because it can be expressed in a short way by stating a recipe or program for its construction. This observation leads to the definition of the information content of a string as the length of the shortest program that computes the string without additional data.

The proposed definition captures the intuitive notion about complexity but it is still unsatisfactory as it makes the amount of information in a string depend on the particular programming language that is used to construct it. It can be shown, however, that for all reasonable choices of programming languages an amount of 'absolute' information can be found that is invariant up to an additive constant. This quantity is called the Kolmogorov complexity of the string.

20 [Li and Vintányi, 91] as cited in [Adriaans, 92]

The absolute amount of information of a binary string x can be defined relative to a Turing machine T and a binary string y as the shortest program p that gives output x on input y:

$$K_T(x|y) = \min\{ |p| : p \in \{0, 1\}^* \text{ and } T(p, y) = x\}$$

Any Turing machine T can safely be replaced by a universal Turing machine U as this does not change any complexity bounds. In the notation the subscript of K is dropped consequently. This leads to the definition of the Kolmogorov complexity of a binary string x as $K(x) = K(x|\varepsilon)$.

2.3.4 A fair probability distribution

The definition of the fair probability distribution according to which the positive examples have to be selected is based on results in complexity theory by Solomonoff, Levin and Li and Vitányi.[21] Before presenting it, I will give a brief overview of the terminology and the results mentioned. A function $w : S \rightarrow [0, 1]$ is called a 'semi-measure' if

$$\sum_{x \in S} w(x) \leq 1$$

It is called a measure or a probability distribution over the space S if this relation is an equation. A function $f : N \rightarrow R$ is enumerable if the set $\{(x, y) \mid y \leq f(x)$ with $x, y \in Q\}$ is recursively enumerable. An enumerable semi-measure μ is called universal if $\mu(x) \geq c\mu'(x)$ for every other enumerable semi-measure μ' and a constant c independent of x. Levin proved the existence of a universal enumerable semi-measure and in the following we fix a universal semi-measure $\mathbf{m}(x)$.

Solomonoff and Levin defined the universal probability $P_U(x)$ of a binary string x as follows:

$$P_U(x) = \sum_{U(p) = x} 2^{-|p|}$$

The following equation proved by Levin relates these concepts and Kolmogorov complexity with each other, basically stating that the likelihood that an arbitrary speaker selects a certain binary string decreases exponentially with the complexity of the string.

$$-\log \mathbf{m}(x) = -\log P_U(x) + O(1) = K(x) + O(1)$$

Li and Vitányi could prove the following completeness result that relates these findings to the concept of pac-learnability: A concept class C is learnable under $\mathbf{m}(x)$ iff C is also learnable under any arbitrary simple distribution $P(x)$, i.e a distribution that is dominated by a recursively enumerable distribution, provided the samples are taken according to $\mathbf{m}(x)$. The definition for the notion of sampling according to $\mathbf{m}$ for any given language is provided by Adriaans:

> Let f be some unique binary coding sequence of a language L. We define a new complexity measure $\mathbf{m_1}$ such that $\mathbf{m_1}(s) = \mathbf{m}(f(s))$. The measure $\mathbf{m_1}$ is a faithful representation of $\mathbf{m}$ provided f is faithful to the complexity of s.

Adriaans assumes that such a faithful coding exists for each language. He also notes that in practical applications it is not possible to actually sample according to $\mathbf{m}$ as it is not constructive. However, the proof of learnability according to $\mathbf{m}$ is in practice a strong indication for

21 [Li and Vintányi, 90] and [Li and Vintányi, 91] as cited in [Adriaans, 92]

learnability in natural situations as it models the linguistic behaviour of a cooperative teacher, who if time is limited will prefer simple examples to complex ones. In this respect we can see some similarity in this approach to other work based on a notion of a minimum description length.

2.3.5 Shallow languages

Adriaans' work is based on CGs. The motivation for this restriction is the belief that CGs have sufficient expressive power to model natural languages. It is for this reason only that CGs provide the starting point on the complexity scale from which further restrictions will be introduced. In this context it is important to recall from paragraph 1.4.2 that, after Adriaans published his results, it was shown that CGs are weakly equivalent to CFGs and therefore the results are transferrable to the more popular sibling.

I have also mentioned before that the complexity bound of a particular machine model is probably not very suited to delineate the class of natural languages. And, in fact, Adriaans introduces two naturalness conditions for the lexicon of CGs that solely depend on Kolmogorov complexity. These naturalness conditions in turn motivate the class of 'shallow' languages that mark the class of languages that can be learnt. To understand the definition of theses languages we need a definition of a 'most complex simple example sentence' of a language first. For every rule r from a grammar G there is a possibly infinite set of sentences for which this rule is used, the example sentences for the rule $ES(r)$. A subset of $ES(r)$ containing only the simplest examples according to their Kolmogorov complexity is $SES(r)$. The union over of the $SES(r)$ for all rules of G is $SES(L)$, where $L = L(G)$ of course. The most complex sentence or sentences in $SES(L)$ form the set of the most complex simple example sentences of a language, $MCSES(L)$. To illustrate these concepts I will give an example for a CG. Note, however, that the complexity of a sentence is not necessarily proportional to its length.

$D = \langle \text{John}, np\rangle, \langle \text{walks}, np\backslash s\rangle, \langle \text{and}, s\backslash s/s\rangle$

$ES(\text{John}) = \{\text{John walks, John walks and John walks, } \ldots\}$

$ES(\text{walks}) = \{\text{John walks, John walks and John walks, } \ldots\}$

$ES(\text{and}) = \{\text{John walks and John walks, John walks and John walks and John walks, } \ldots\}$

$SES(\text{John}) = \{\text{John walks}\}$

$SES(\text{walks}) = \{\text{John walks}\}$

$SES(\text{and}) = \{\text{John walks and John walks}\}$

$SES(L) = \{\text{John walks, John walks and John walks}\}$

$MCSES(L) = \{\text{John walks and John walks}\}$

Given these sets, Adriaans defines shallow languages as follows:

> A language L with a grammar G is shallow if $K(MCSES(L)) = O(\log K(G))$.

A strong supposition that natural languages are in fact shallow and thus learnable can be derived from a simple calculation. A native speaker will have passive knowledge of at least 30.000 words, thus a lower bound of the size of the grammar (which, of course, includes the lexicon) is 30.000 or approximately 2^{15}. This means that a natural language would not be shallow only if it had a grammatical construct that could not be exemplified in a sentence with less than 16 words. It is interesting though that most programming languages with their simple grammars but deeply nested syntactic structures are not shallow. This implies that there is no

division line in the Chomsky hierarchy for shallow grammars because their measure of complexity is orthogonal to the one employed by Chomsky.

2.3.6 Learnable Abstract Syntactic Behaviour

In this paragraph the definitions and results from the previous one are combined to formally define the class of learnable languages and to identify under which conditions effective learnability is possible. The first definition is concerned with the environment of the learning algorithm, i.e. the teacher. An abstract speaker is defined as quintuple <*D*, *L*, *S*, E_P, *O*> with:

- *D*, a lexicon set.
- *L*, a language which is a subset of the Kleene closure D* of D.
- *S*, the set of true sentences of *L*.[22]
- E_P, an examples routine producing an element of *S* according to the probability distribution *P* over *L*.
- *O*, an oracle routine that tells us whether or not a certain element of *D** is member of *L* and or *S*.

Based on this model Adriaans defines the effective learnability of a language as follows:

> Suppose we have an abstract speaker *A* and a learning algorithm *LA* that given an accuracy parameter α and a confidence parameter β, asks a number of examples from *L*, calls the Oracle a number of times and constructs a hypothesis. This hypothesis has the form of the characteristic function of a language $H : D^* \rightarrow \{0, 1\}$. We identify this language as L_H.
>
> A class of languages **L** is pac-learnable if we have for every $L \in \mathbf{L}$ and every probability distribution *P* that: $P(L \Delta L_H > \alpha) \leq \beta$, where $L \Delta L_H$ is the symmetric difference between *L* and L_H. A class of languages is effectively learnable if *LA* runs in polynomial time and calls E_P and *O*() a polynomial number of times. Polynomial means here polynomial in $1/\alpha$ and $1/\beta$ and the length of the syntax to be learned. We have Learnable Abstract Syntactic Behaviour if there is an effective algorithm to learn the syntactic behaviour of an abstract speaker *A*.

Note that this, even though it is a clear definition of the meaning of learnability, does neither make the class of learnable languages explicit nor does it incorporate the notion of a cooperative teacher. It does, however, with the distribution *P* provide a means to characterize the teacher's behaviour. Hence, the abstract speaker is replaced by a cooperative abstract speaker by introducing the following restrictions to its definition:

- The language *L* is shallow.
- The example routine is $E_{\mathbf{m}}$, i.e. it samples according to **m**.

As the effective learnability of a language is defined in terms of the existence of an appropriate learning algorithm the final missing piece to prove that natural languages are effectively learnable is the algorithm. Adriaans presents such an algorithm that learns a categorial grammar

22 Adriaans is concerned with semantic as well as syntactic learning. He assumes some mapping $SEM : L \rightarrow \{0, 1\}$.

under the conditions specified before[23] and the Emile algorithm that I will present in the next chapter is a descendant of this algorithm.

2.3.7 Summary

The subject of cooperative language learning is Learnable Abstract Syntactic Behaviour. It requires a teacher that provides positive examples and can answer membership queries. Furthermore, the teacher's behaviour, characterized by the probability distribution, must be cooperative, i.e. fair and methodological. The distribution **m** provides a borderline case for such behaviour. If the language is shallow then there is a learning algorithm which with a high probability but not absolute certainty identifies the language almost perfectly after a reasonable (polynomially bounded) amount of time.

23 [Adriaans, 92], pp. 133–144. Note that the name EMILE is also used for a semantic learning algorithm that Adriaans presents in his dissertation after the syntactic learning algorithm. This algorithm is not related to the EMILE algorithm I am considering. Further, the name EMILE is actually an acronym for 'Entity Modelling Intelligent Learning Engine' which is merely a hint at the origins of it.

Chapter

3 The Emile algorithm

In the last section of the previous chapter I described a framework for cooperative learning and cited Adriaans' results concerning learnability. These results depend on the existence of an appropriate algorithm. In this chapter I will first explain some ideas and observations that are necessary for the construction of such an algorithm. I will also explicate some concepts introduced in previous chapters. However, I will not present the actual algorithm that Adriaans used to prove the effective learnability of natural languages because its very similar to the Emile algorithm and I am concerned that it would confuse more than help the reader understand the Emile algorithm. In section 3.2 I will present the Emile algorithm using a simple language as an example. In this context it is important to note that the original description of the algorithm[1] leaves certain details open and the algorithm I present is my interpretation of Adriaans' and Knobbe's paper. Following the informal description of the algorithm I will outline an efficient implementation in section 3.3. I will examine the data structures involved and estimate the worst case running time. This chapter concludes with a description of some experiments performed with the implementation and an evaluation of the results which will suggest certain improvements that are the subject matter of the remaining part of the thesis.

3.1 A syntactic learning algorithm

3.1.1 Learning with examples and an oracle

The syntactic learning algorithm is designed in the pac-learning framework and therefore it is obvious that it will utilise an example and an oracle routine. The framework does, however, not limit the way in which the algorithm uses the routines and how hypotheses are formed. In fact, it does not even require the algorithm to search for a most general hypothesis or to explore the hypotheses space. I will discuss these questions in relation to the problem of learning finite functions. While this domain is undoubtedly much simpler than the domain of languages it is suitable to exemplify the design of the actual language learning algorithm.

In line with the definitions of the abstract speakers in the previous chapter, a functional speaker is defined as a 4-tuple <U, f, E_P, O> with:

- $U = D \times R$, a universe of discourse. D being the domain of the function and R its range.
- $f : D \rightarrow R$, the target function to be learned.
- E_P, an examples routine that produces statements of the form $f(v) = u$ for arbitrary elements $\langle u, v \rangle \in U$ according to the probability distribution P. Note that this example routine provides positive examples only.
- O, an oracle routine that returns 1 if a given pair $\langle u, v \rangle$ is member of f and 0 otherwise.

1 [Adriaans and Knobbe, 96]

The first algorithm 'learns' a finite function on the basis of positive examples without making use of the oracle. Note that in this and the following algorithm the variable f_{part} contains the partial knowledge about the function and f_{comp} knowledge about the complement of the function.

```
f_part = {}
for x = 1 to n do
    '(f(v)=u)' = EP()
    f_part = f_part ∪ {<u,v>}
```

Figure 3. Algorithm 1, learning finite functions by example.

This is probably the most simple conceivable learning algorithm and it does not go beyond even the most general definition of 'learning by example' presented in paragraph 2.1.3. The oracle is not used, but provided E_P samples over the entire universe of discourse the algorithm will eventually get every example necessary. Therefore, the effectiveness of the learning process and the completeness of the result depend solely on *P.*

Now consider the algorithm 2. It starts with initial knowledge about the function K_f, the domain of the function K_D and the range of the function K_R which must not be empty. This algorithm

```
f_part = Kf
f_comp = {}
domain = KD
range = KR
foreach p in {x | x ∈ domain} do
    foreach q in {y | y ∈ range} do
        if (not(<p,q> ∈ f_part) and not(<p,q> ∈ f_comp) then
            if O(<p,q>) == 1 then
                f_part = f_part ∪ {<p,q>}
            else
                f_comp = f_comp ∪ {<p,q>}
```

Figure 4. Algorithm 2, learning finite functions with initial knowledge and an oracle.

uses the initial knowledge to delimit the search space and within this 'window' it obtains a locally complete learning result. In this version the space for hypotheses that has to be explored by means of the oracle is $K_D \times K_R$. Using some knowledge about the domain, a simple change can be introduced that reduces the size of the hypotheses space considerably, namely only checking those x for which the following condition holds: $\forall y : \langle x, y\rangle \notin f_{part} \cup f_{comp}$. For other domains similar improvements are conceivable.

Algorithm 3 is a combination of these two basic algorithms. It uses 'passive' learning by example to explore the universe of discourse and 'active' learning by oracle to get local completeness results. It is thus dependent on the distribution *P* but effectiveness can be increased if the algorithm prunes the hypotheses space by incorporating knowledge about the domain. It is an obvious weakness of the algorithm that the number of examples that are asked from the examples routine is determined by a constant parameter but it is generally not possible to estimate

```
f_part = {}
f_comp = {}
domain= {}
range = {}
for x = 1 to n do
    '(f(v)=u)' = EP()
    f_part = f_part ∪ {<v,u>}
    domain = domain ∪ {v}
    range = range ∪ {u}
foreach p in {x | x ∈ domain} do
    foreach q in {y | y ∈ range} do
        if (not(<p,q> ∈ f_part) and not(<p,q> ∈ f_comp)) then
            if O(<p,q>) == 1 then
                f_part = f_part ∪ {<p,q>}
            else
                f_comp = f_comp ∪ {<p,q>}
```

Figure 5. Algorithm 3, learning finite functions.

the number of examples needed to learn if the size and complexity of the universe of discourse are not known in advance. And, unfortunately, the algorithm uses the examples routine to explore the universe of discourse.

3.1.2 The Lambek calculus and partial information

After outlining the general flow of the algorithm I will now return to the intended domain, i.e. the Lambek calculus, and explain how hypotheses are formed. The standard Lambek calculus does allow for the deduction of set inclusions.[2] If, for example, the learning algorithm discovers the proof $f : B / B \bullet A \rightarrow B$, it can use the calculus to derive that A is a subset of B:

$$\dfrac{\dfrac{\dfrac{f : B/B \bullet A \rightarrow B}{f^* : B/B \rightarrow B/A} \quad \dfrac{\lambda_B : I \bullet B \rightarrow B}{\lambda_B^* : I \rightarrow B/B}}{\lambda_B^* f^* : I \rightarrow B/A}}{\dfrac{(\lambda_B^* f^*)^+ : I \bullet A \rightarrow B \qquad \lambda_A^{-1} : A \rightarrow I \bullet A}{\lambda_A^{-1}(\lambda_B^* f^*)^+ : A \rightarrow B}}$$

However, in learning situations it is often necessary to create new sets. Consider the following examples from a language that contains sentences consisting of a name and an intransitive verb.

a : {Mary} • {walks} → S

b : {John} • {walks} → S

2 A proof like $f : A \rightarrow B$ can be interpreted such that A is a subset of B.

If the algorithm discovers the proofs *a* and *b* it would be reasonable to combine “Mary” and “John” into a new type and then express the type of the verb in terms of this type.

c : {John, Mary} • {walks} → S

d : {walks} → {John, Mary} \ S

Both goals can be achieved with the help of the rules of the extended Lambek calculus presented in paragraph 1.4.4. It can be shown that c = $[a^*, b^*]^+$ and d = $*([a^*, b^*]^+)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{a : \{\text{Mary}\} \bullet \{\text{walks}\} \rightarrow S}{a^* : \{\text{Mary}\} \rightarrow S / \{\text{walks}\}} \quad \dfrac{b : \{\text{John}\} \bullet \{\text{walks}\} \rightarrow S}{b^* : \{\text{John}\} \rightarrow S / \{\text{walks}\}}}{[a^*, b^*]: \{\text{Mary}\} \vee \{\text{John}\} \rightarrow S / \{\text{walks}\}}}{[a^*, b^*]^+ : (\{\text{Mary}\} \vee \{\text{John}\}) \bullet \{\text{walks}\} \rightarrow S}}{*([a^*, b^*]^+) : \{\text{walks}\} \rightarrow (\{\text{Mary}\} \vee \{\text{John}\}) \backslash S}$$

This covers the operations which are needed to evaluate the positive examples. I have stated above that it would be reasonable to combine “John” and “Mary” into a type. While this is true for this particular case it is in general a too optimistic assumption as can be illustrated by the following example:

a : ({He} • {sees}) • {John} → S

b : ({He} • {sees}) • {flowers} → S

After processing *a* and *b*, a hypothesis like *c* that “John” and “flowers” belong to the same type will be formed. But instead of blindly creating a new type as described above, the algorithm uses the oracle to check the expression in different contexts obtained from other examples.

c : ({He} • {sees}) • ({flowers} ∨ {John}) → S

d : {John} • {walks} → S

e : {Flowers} • {walks} → S

The negative result for *e* falsifies the hypothesis and leads to the addition of a quite different proof *f* instead of *c*.

f : {Flowers} • {walks} → ¬S

Combining these operations with the learning algorithm presented in the previous paragraph yields an outline of the actual Emile algorithm. The algorithm receives a set of positive examples, forms hypotheses about set inclusions and new types, i.e. sets of expressions, and verifies these hypotheses with the help of an oracle.

3.2 Emile step-by-step

3.2.1 Selecting example sentences

After characterizing the general structure of the algorithm and describing the basic operations involved, I present in this section the Emile algorithm by means of an example. The target lan-

guage is a small subset of English and it is assumed that the examples routine has chosen the following two sentences:

(1) Mary walks.
(2) John loves Mary.

It is assumed that these example sentences are chosen according to a distribution recursively dominated by **m** and that they span a universe of discourse that is roughly equivalent to the subset of English which is considered.

3.2.2 First order explosion

The example sentences have to be brought into a form such that the hypothesis space can be explored effectively. The most general strategy would be to assume that each word could be of the same type as each other word regardless of its position in the sentences in the sample. In this case the algorithm would have to explore an number of hypotheses exponential in the length of the sentences. Given that s is the number of sentences in the sample, n_i is the number of words in sentence i, n is the maximum length of a sentence and l is an assumed maximum length of a sentence in the language, the size of the hypothesis space would be:

$$\sum_{i=1}^{l} (sn)^i = O((sn)^l)$$

The number of words, sn, is only an upper bound as some words usually occur in more than one example sentence. It would, of course, be possible to limit the search to sentences not longer than the longest sentence in the sample, i.e. $l = \max\{n_i\}$, but this seems to be too optimistic.

The Emile algorithm employs a different strategy and works with contexts and expressions. It makes use of rules to divide a sentence into a context and an expression, that bear some similarity to the transformation rules in the Lambek calculus, i.e. equation (1.13) on page 16. Consider the following example:

(1) John loves Mary $\rightarrow S$
(2) S / loves Mary $\rightarrow$ John

The sentence in rule (1) is split into a context, on the left of the arrow in rule (2), and an expression on the right of it. Alternatively, this operation could be described as follows: A rule of the form $\alpha\beta\gamma \rightarrow S$, with $|\alpha| \geq 0$, $|\gamma| \geq 0$ and $|\beta| > 0$, can be rewritten as $\alpha\backslash S/\gamma \rightarrow \beta$. The Emile algorithm forms the so-called ‘first order explosion’ by generating all possible pairs of contexts and expressions from the sentence in the sample.

S / loves Mary $\rightarrow$ John	John loves $\backslash$ $S \rightarrow$ Mary
S / Mary $\rightarrow$ John loves	S / walks $\rightarrow$ Mary
$S \rightarrow$ John loves Mary	$S \rightarrow$ Mary walks
John $\backslash$ S / Mary $\rightarrow$ loves	Mary $\backslash$ $S \rightarrow$ walks

For a sentence of length n_i there are n_i expressions of length 1, (n_i–1) expressions of length 2, etc. resulting in the following number of expression/context pairs for a single sentence:

$$\sum_{i=1}^{n_i} (n_i - i + 1) = \frac{1}{2} n_i (n_i + 1)$$

Therefore, the upper bound for the number of contexts and the number of expressions constructible from the sentences in the sample is:

$$\sum_{i=1}^{s} \frac{1}{2} n_i (n_i + 1) = O(sn^2)$$

This is only an upper bound insofar as the number of contexts and expressions will be smaller because the same expression or context will occur in several sentences in the sample, especially for languages with a small lexicon.

It is assumed that each expression might be inserted into every context and therefore the expressions on one side and the contexts on the other span the hypotheses space. Therefore, the following upper bound for size of the hypotheses space holds:

$$\left(\sum_{i=1}^{s} \frac{1}{2} n_i (n_i + 1) \right)^2 = O(s^2 n^4)$$

If n_m is the length of the longest sentence in the sample, the algorithm will explore sentence with a length of up to $2n_m$–1, i.e. the longest sentence is inserted into a context resulting from a division of the longest sentence into the context and an expression consisting of one word only. It must be noted that by limiting the maximum length of a potential sentence, the complexity bound of the exhaustive search algorithm becomes polynomial. However, this is only of theoretical interest as the order will be prohibitively large for even the smallest samples. The example has 2 sentences, 4 different words, 8 different contexts and 8 different expressions. Hence, the length of the longest sentence to be checked is 5 and the hypotheses spaces have the following sizes:

$$\sum_{i=1}^{5} 4^i = 1364 \quad \text{for the exhaustive search and} \quad 8 \cdot 8 = 64 \quad \text{for Emile}$$

The hypotheses space, or, as I will call it in the next paragraphs, the substitution matrix, for the example is shown in figure 6.

3.2.3 Enriched first order explosion

The next stage in the Emile algorithm is to explore the substitution matrix with the help of an oracle. Figure 6 shows the results and every context/expression combination that was rejected by the oracle is marked with a dot, while the others contain a letter and are shaded. Entries marked with an “S” indicate that the resulting sentence was present in the sample already. Note that certain combinations, such as “S / walks → John” and “John \ S → walks,” form the same sentence and the answer obtained from the oracle for one of them can be used for the other. The set of combinations that from a sentence accepted by the oracle is called ‘enriched first order explosion’ and is in the example constituted by the following rules:

	John	John loves	John loves Mary	loves	loves Mary	Mary	Mary walks	walks
S / loves Mary	S	•	•	•	•	O	•	•
S / Mary	•	S	•	•	•	•	•	•
S	•	•	S	•	•	•	S	•
John \ *S* / Mary	•	•	•	S	•	•	•	•
John \ *S*	•	•	•	•	S	•	•	O
John loves \ *S*	O	•	•	•	•	S	•	(•)
S / walks	O	•	•	•	•	S	•	•
Mary \ *S*	•	•	•	•	S	•	•	S

Figure 6. The substitution matrix.

S / loves Mary → John
S / Mary → John loves
S → John loves Mary
John \ *S* / Mary → loves
S / loves Mary → Mary
John loves \ *S* → John
Mary \ *S* → loves Mary
John loves \ *S* → Mary
S / walks → Mary
S → Mary walks
Mary \ *S* → walks
John \ *S* → walks
S / walks → John

Note that the sentence “John loves walks” is not considered to be part of the language, demonstrating that the oracle can apply any criterion to determine whether the sentence is correct. It would even be conceivable that the oracle only accepts sentences that are ‘true’ according to some interpretation. This, however, will often result in unlearnability because the underlying model is not expressible in a shallow context-free language. Therefore, in practical applications any incorporation of semantic criteria into the oracle, and that includes for example a distinction between animate and inanimate nouns to reject sentences like “Water sleeps,” must be considered carefully to maintain learnability.

3.2.4 Clustering

According to the framework in paragraph 3.1.1, the algorithm could halt at this stage as it has constructed a valid model for the sentences in the sample, one that even includes some generalisations. However, it is the goal of the Emile algorithm to find a CFG that is as general as possible and therefore it tries to find sets of expressions that belong to the same type according to the description in paragraph 3.1.2. It is interesting to note that the Emile algorithm queries the oracle first and then determines the types, instead of forming a hypothesis about a type and then validating this hypothesis with the help of the oracle.

Returning to the example, the following rules indicate a type:

S / loves Mary → John
John loves \ *S* → John
S / walks → John
S / loves Mary → Mary
John loves \ *S* → Mary
S / walks → Mary

The expressions “John” and “Mary” can be substituted in exactly the same set of contexts and therefore the algorithm introduces a new symbol *A* to describe this type. The rules above are discarded and replaced by the following set:[3]

A → John | Mary
S / loves Mary → *A*
John loves \ *S* → *A*
S / walks → *A*

It is possible to visualise the types in the substitution matrix by reordering the rows and columns such that rectangles of valid substitutions can be found along the diagonal of the matrix.

	John	Mary	John loves	John loves Mary	Mary walks	loves	loves Mary	walks
S / loves Mary	S	O	•	•	•	•	•	•
John loves \ *S*	O	S	•	•	•	•	•	(•)
S / walks	O	S	•	•	•	•	•	•
S / Mary	•	•	S	•	•	•	•	•
S	•	•	•	S	S	•	•	•
John \ *S* / Mary	•	•	•	•	•	S	•	•
John \ *S*	•	•	•	•	•	•	S	O
Mary \ *S*	•	•	•	•	•	•	O	S

Figure 7. The reordered substitution matrix.

In most practical cases it is impossible to achieve such a reordering though. Consider the case of the sentence “John loves walks” again. The word “walks” is homonymous as this string of characters can represent two different words, namely the third person singular (present) of the verb “walk” and the plural of the noun “walk.” If some sort of semantics is incorporated another problem occurs. In most languages polysemous words, i.e. words with several meanings, share one of their meaning with another word. For example “mature” and “ripe” are interchangeable when applied to fruit, but only “mature” can be used in conjunction with animals or persons.

The nature of these problems is best expressed with graph theory. The algorithm would ideally require a bipartite matching between the sets of expressions and contexts as shown in figure 8. In practical applications, however, it is confronted with less ideal situations. The situation depicted in figure 9a turns out to be unproblematic for the algorithm in its current version as it defines types on the basis of expressions that occur in the same context; whether or not some contexts are valid for expressions of a different type is irrelevant. The algorithm will, in the clustering stage, define two types and for each context in the intersection of the sets it will construct two rules, one stating that expressions of type *A* can be inserted and the second one stating that the context is also valid for expressions of type *B*.

3 For the sake of brevity I will use a bar on the right-hand side to indicate alternatives.

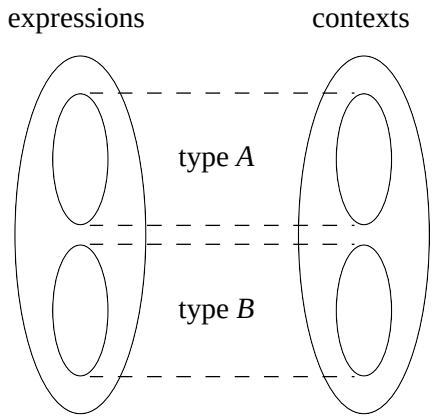


Figure 8. A bipartite matching between expressions and contexts.

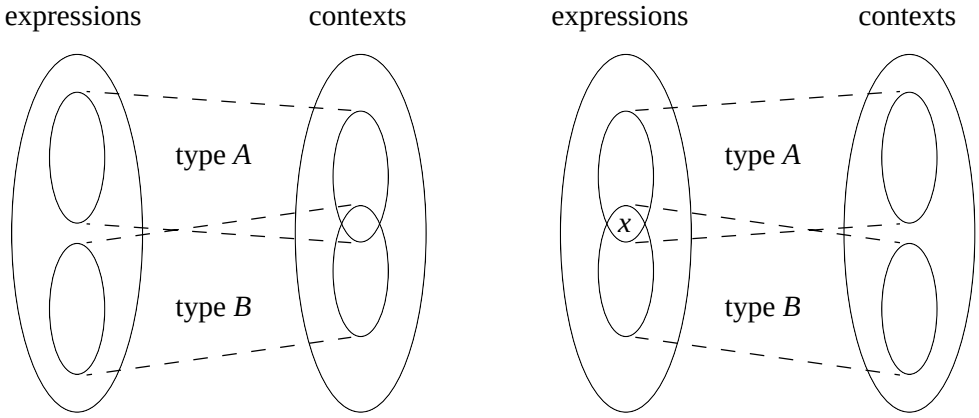


Figure 9a and b. Overlapping sets of contexts and expressions.

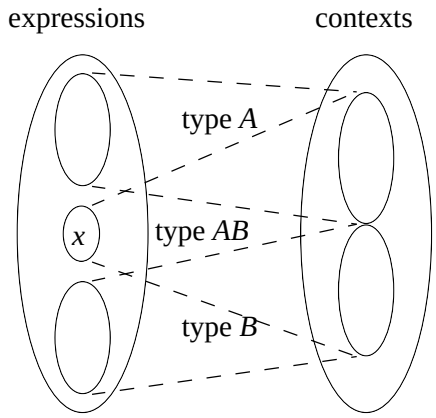


Figure 10. Resolving overlapping sets of expressions.

Because of homonymy and polysemy for example, an expression *x* can conceptually belong to more than one type as shown in figure 9b. The Emile algorithm in its current version solves this problem by simply ignoring it, i.e. it resolutely searches for expressions that can be substituted in exactly the same set of contexts. If an expression like “walks” can be substituted in contexts that are valid for, say, certain intransitive verbs in third person singular present tense as well as in contexts that take certain noun phrases, it is assigned to neither of the corresponding types

but to a new type that can be substituted in the union of the two context sets.[4] This obviously results in overlapping contexts but as explained before this poses no problem for the algorithm.

After the clustering stage the rule set for the example consists of the following rules:

S / loves Mary $\rightarrow A$
John loves \ $S \rightarrow A$
S / walks $\rightarrow A$
S / Mary $\rightarrow B$
$S \rightarrow C$
John \ S / Mary $\rightarrow D$
John \ $S \rightarrow E$
Mary \ $S \rightarrow E$

$A \rightarrow$ John | Mary
$B \rightarrow$ John loves
$C \rightarrow$ John loves Mary | Mary walks
$D \rightarrow$ loves
$E \rightarrow$ loves Mary | walks

Note that the rule set is partitioned into basic lexicon rules such as "$A \rightarrow$ Mary" and more complex rules like "S / loves Mary" and "$B \rightarrow$ John loves."

3.2.5 Generalisation

In the generalisation stage the algorithm transforms the complex rules so that they are expressed in terms of the lexicon rules. Each word in a complex rule is replaced by the type it belongs to. For example the rule "S / loves Mary $\rightarrow A$" is changed to "S / $D\,A \rightarrow A$." This transformation is legal as the types consist of expressions that can be replaced for each other in any context. It is a generalisation because a set of expressions, some of which can consist of several words, is substituted for a single word.

Finally, all rules that contain contexts are transformed with a rule similar to the ones in equation (1.14) on page 16. For the example this results in the following set of context-free rules:

(1) $S \rightarrow A\,D\,A$
(2) $S \rightarrow B\,A$
(3) $S \rightarrow C$
(4) $S \rightarrow A\,E$
(5) $B \rightarrow A\,D$
(6) $C \rightarrow A\,D\,A \mid A\,E$
(7) $E \rightarrow D\,A \mid$ walks
(8) $A \rightarrow$ John | Mary
(9) $D \rightarrow$ loves

The subset consisting of rules the in lines (4), (7), (8) and (9) corresponds to an intuitive grammar for the language. This becomes more obvious if the names are replaced as follows: A – NP, E – VP and D – VT:

(4) $S \rightarrow$ NP VP
(7) VP $\rightarrow$ VT NP | *walks*
(8) NP $\rightarrow$ *John* | *Mary*
(9) VT $\rightarrow$ *loves*

4 The syntactic learning algorithm in [Adriaans, 92] detects such conditions and disambiguates these expressions with the help of further questions to the oracle.

The rule set also contains other subsets that generate the entire language. Therefore, each sentence is usually assigned a multitude of parse-trees with very different structures. I will discuss the consequences in section 3.5. For the moment it suffices to note that the algorithm does in fact learn a valid grammar for the example language.

3.3 An efficient implementation

3.3.1 The implementation environment

While many of the properties of the Emile algorithm can be analysed on the basis of the description in the last section and the underlying models, such as the Lambek calculus, it is important, especially for the evaluation of the usefulness in practical applications, to have an implementation of the algorithm with which experiments can be carried out. A seemingly obvious choice of an implementation environment would have been Prolog. However, the algorithm is not very complicated; it consists mainly of the application of a few simple transformations on potentially vast amounts of data. Furthermore, the number of data types involved is limited and they have a rather simple structure. For these reasons I decided to implement the algorithm in an imperative language. Because I believe that principles such as abstraction and encapsulation result in a clearer and generally superior design I considered object-oriented environments only.

My environment of choice was Java,[5] but at the time I began with the implementation the class libraries, or packages as they are called in this environment, were still quite immature and incomplete. Another problem was the execution speed as no 'just-in-time' compilers were available. I therefore decided on the Objective-C language,[6] a language that, despite its Smalltalk influenced syntax, is very similar to Java. In fact, it is identical if one considers only those aspects that are used in my implementation of the Emile algorithm. As a library I chose the Foundation classes from NeXT Software's Openstep environment.[7] These classes include standard collections such as arrays, dictionaries and sets and are designed according to standard design patterns such as Iterator, Memento, Strategy and Visitor.[8]

In anticipation of the estimates of the running time for the algorithm some remarks are necessary. Retrieving an object by index from an array and all operations at the end of an array can be performed in $O(1)$. Insertions and deletions anywhere else require $\Theta(n)$. Arrays can grow and shrink dynamically when necessary but it can be shown with amortized analysis[9] that performance does not deteriorate under these circumstances. Dictionaries and sets are based on hash tables and thus all operations can be performed in $O(1)$ if the table contains at least approximately 20% free entries. The latter is ensured by dynamically resizing these structures. Another important point is the distinction between mutable and immutable collections, the latter of which cannot be changed once they are initialised with a certain set of elements. This

5 [Arnold and Gosling, 96]

6 [Cox and Novobilski, 91]

7 NeXT Software was acquired by Apple Computer in a friendly takeover and the Foundation classes will be part of the forthcoming operating system code-named 'Rhapsody.' See <http://www.apple.com> for further details.

8 Pattern names according to [Gamma *et al.*, 95].

9 [Cormen, Leiserson and Rivest, 89], pp. 367–374

allows for further improvements of running time. Consider an array A with n elements of which a subarray B ranging from index i to j, with $0 \leq i \leq j < n$, is to be created. Under normal circumstances $j - i$ elements (or references to them) have to be copied into a new array resulting in running time and space to be proportional to the length of the subarray. If array A and B are known to be immutable, an object can be created for array B that merely contains the offset into array A and the length of the subarray. This operation is obviously $O(1)$. I will not always mention the actual flavour of the collection used, but for running time estimates it is assumed that immutable collections are used where possible. Finally, the array class provides a sorting algorithm and I will assume that its running time is bounded by $O(n \log n)$.

3.3.2 The example and oracle routines

The implementation of Emile has no examples routine. It simply reads a number of sentences from a text file. This makes it easy to experiment with a variety of tools to select an appropriate sample, a task that is more complicated than it might appear. Adriaans arrives at the learnability results by sampling according to a decidedly non-constructive distribution and assumes that this distribution captures the behaviour of a cooperative teacher. I will describe in section 3.4 how samples were drawn on an ad hoc basis for various experiments and why these samples were deemed fair in Adriaans' sense.

The implementation defines an abstract interface for a potential oracle class which makes it possible to switch easily between various implementations. The concrete implementation may vary from experiment to experiment. Recalling the size of the substitution matrix it is clear that at no point the experimenter has to answer the questions manually. However, the implementation of such an oracle class, when it becomes necessary, is obviously trivial.

3.3.3 Model objects

Before turning to the implementation of the algorithm itself I will describe some classes that are used to model the data. The implementation uses two different representations for sequences of words, either an array of string objects or a single string object in which the individual words are separated by spaces. The latter representation is used less often and is mainly justified by efficiency reasons.

There is a context class, instances of which represent the contexts generated by the first order explosion. Instances of it store both sides of the context and provide an efficient method that takes a sequence of words, inserts them into the context and returns the resulting sequence. The expression class represents expressions in a similar way. However, expression objects not only contain the sequence of words by which they are defined but they also provide a means to store information about which contexts they are valid in and which expression type they belong to.

As stated before a type is merely a set of expressions. Therefore, instances of the expression type class mainly consist of a set containing all expression objects that constitute the type. They also allow for the designation of one of the expressions as a representative for the whole set. Furthermore, expression types can be assigned an arbitrary string as a name and have a method that generates a sequence of unique names for this purpose.

Instances of the rule class represent the context-free rules that are constructed in the generalisation stage. They store a left-hand side, which is always an expression type object, and a right-

hand side which is either an array of expression type objects or, in the case of lexicon rules, a string object representing the corresponding word. It can be determined by other objects whether or not a rule is a lexicon rule. Rule objects can also create two different string representation of themselves in the DCG formalism. The first one consists of the names of the expression types and bracketed words, while the second version is enriched with parameters that build a structured representation of the parse tree when they are used in a Prolog interpreter. As an example consider the following rules:

(1) $E \rightarrow D\ A$

(2) $D \rightarrow$ loves

The DCG rules created from them are:

(1) `e --> d, a.` and `e(e(X0, X1)) --> d(X0), a(X1).`

(2) `d --> [loves]` and `d(d(loves)) --> [loves].`

All model classes implement certain methods required by the Foundation classes. They can usually calculate a hash value which is used by collection classes such as dictionary and set. The model classes also have methods with which instances can be tested for equality or ranked according to some order. Finally, the context, expression and expression type classes can interact with a visitor object which is used by a persistency mechanism that can archive and restore complete object graphs.

All operations on the model classes that are constructive, i.e. create a new object, have a running time of $O(n)$ according to an intuitive notion of the length of data involved. Exceptions to this rule are noted individually. All operations that query attributes are performed in $O(1)$ unless stated otherwise.

3.3.4 The Emile object

The algorithm is encapsulated in a class as well. This class has a couple of state variables such as the sets of expressions, contexts and rules. It provides a single method for each stage of the algorithm and keeps track which stages have been completed. It also allows for archiving in between the stages so that certain stages can be run with different parameters without having to go through all preceding stages. The object is initialised with an array containing the sample.

3.3.5 Stage 1: First order explosion

The task of the first order explosion is to find all possible combinations of contexts and expressions for the sentences in the sample. The algorithm simple iterates over all sentences. For each sentence it iterates over all possible indices at which the expression can start and for each index over the possible lengths. For each such combination, expression and context objects are created and stored in sets. If the new expression compares as equal to an existing one, i.e. consists of the same words, it is discarded and the previously created one is used. The context is then added to the expression's set of valid contexts.

As explained in paragraph 3.2.2 the number of context/expression pairs is $O(sn^2)$, the creation of each pair takes $O(n)$ and the check whether the expression is in the set of previously created ones takes $O(1)$. The running time for stage 1 is therefore bounded by $O(sn^3)$.

```
forall s in {x | x ∈ sentences} do
    l = s.length
    for i = 0 to l do
        for j = 1 to l - 1 do
            c = new Context
            e = new Expression
            c.leftSide = s.subarray([0, i])
            e.words = s.subarray([i, j])
            c.rightSide = s.subarray([i+j, l-(i+j)])
            contexts.add(c)
            if expressions.contains(e) then
                e = expressions.get(e)
            else
                expressions.add(e)
            e.contexts.add(c)
```

Figure 11. The Emile algorithm, stage 1

3.3.6 Stage 2: Enriched first order explosion

The implementation of the enriched first order explosion is fairly straightforward. In an outer loop the algorithm iterates over all contexts and in an inner loop it iterates over all expressions. Each combination is tested for validity by first checking whether the resulting sentence has been judged by the oracle before and if not by querying the oracle. In the latter case the answer is stored in a dictionary with the sentence as key and two designated objects, one for positive and one for negative answers, as value. If the answer is positive the context is added to the expression's set of contexts. Note that the answer dictionary is initialised with the insertions that are known to be valid from the example sentences. This is not shown in the code.

```
foreach c in {x | x ∈ contexts} do
    foreach e in {x | x ∈ expressions} do
        s = c.insertExpression(e)
        if answers.containsKey(s) then
            a = answers.valueForKey(s)
        else
            a = oracle.check(s)
            answers.setValueForKey(a, s)
        if a == 'OK' then
            e.contexts.add(c)
```

Figure 12. The Emile algorithm, stage 2

The size of the substitution matrix is $O(s^2n^4)$. Ignoring the oracle, the most expensive operation is the insertion of the expression into the contexts which is linear in the length of the resulting sentence. Thus, the running time for stage 2 is bounded by $O(s^2n^5)$.

The worst case result for the following stages is a substitution matrix that contains more contexts than expressions and one expression which is valid in all contexts and all other expressions are valid in different permutations of all but one context. This case is illustrated in figure 13.

	ex 1	ex 2	ex 3	…	ex n_e-1	ex n_e
context 1	O	•	O		O	O
context 2	O	O	•		O	O
context 3	O	O	O		O	O
…						
context $n_e - 2$	O	O	O		•	O
context $n_e - 1$	O	O	O		O	•
…						
context n_c	O	O	O		O	O

Figure 13. Worst case substitution matrix

3.3.7 Stage 3: Clustering

At this stage the exploration of the hypotheses space is completed and stored in each expressions object is a list of contexts in which this expressions can be inserted. To determine the expression types it is now necessary to compare the corresponding sets to each other and combine those expressions into a type that are valid in the same contexts. The algorithm iterates over all expressions. For each expression it first checks whether the set of contexts that the current expression is valid in is equal to the set of contexts that the designated reference expression from any known type is valid in. In this case the expression is added to the existing type. Otherwise a new type is created and the current expression is made its representative.

```
foreach e in { x | x ∈ expressions } do
    foreach t in { x | x ∈ types } do
        r = t.representative
        if e.contexts == r.contexts then
            t.extension.add(e)
            e.type = t
            continue with next e
    t = new Type
    t.representative = e
    t.extension.add(e)
    e.type = t
    types.add(t)
```

Figure 14. The Emile algorithm, stage 3

The running time of this stage for the worst case is $O(s^3n^6)$. There are $O(sn^2)$ expressions and consequently at most $O(sn^2)$ types, which means that the body of the inner loop can be reached $O(s^2n^4)$ times. Whether two sets are equal can be determined in $O(m)$ where m is the size of the smaller set. In the worst case the size of the largest smaller set is the number of contexts minus one and thus the comparison of the sets in the body of the inner loop can be performed in $O(sn^2)$.

3.3.8 Stage 4: Generalisation

For each expression type there are two classes of rules that have to be created. Rules of the first class state that the context forms a valid sentence with the expressions of this type and the second class of rules represents the fact that various expressions belong to the expression type. For example, for an expression type A consisting of the expressions x, y and z which can be inserted into the contexts $a\backslash S/b$ and S/c the following rules are created:

(1) $S \rightarrow a\ A\ b$, $S \rightarrow A\ c$
(2) $A \rightarrow x$, $A \rightarrow y$, $A \rightarrow z$

At the same time all words must be replaced by the type they belong to. For this reason a dictionary is created that maps each word onto its type. Furthermore, a dummy expression type representing the start symbol is created. Both operations are trivial and not shown below.

To create rules of the first class the algorithm iterates over all types and for each type over all contexts that expressions of this type are valid in. An new rule is created with the expression type representing the start symbol on the left-hand side. The right-hand side of the rule is made up by the sequence of expression types for the words on the left side of the context, followed by the current expression type, followed by the sequence of expression type for the words on the right side of the contexts

```
foreach t in { x | x  ∈ types } do
    foreach c in { x | x ∈ t.representative.contexts } do
        rule = new Rule
        rule.lhs = startsymbol
        rule.rhs = {}
        foreach w in { x | x ∈ c.leftSide } do
            wt = wordtypes.valueForKey(w)
            rule.rhs.append(wt)
        rule.rhs.append(t)
        foreach w in { x | x ∈ c.rightSide } do
            wt = wordtypes.valueForKey(w)
            rule.rhs.append(wt)
```

Figure 15. The Emile algorithm, stage 4a

Rules of the second class are created in a similar way. Again, the algorithm iterates over all expression types. In an inner loop it iterates over the extension of this type, i.e. all expressions that belong to it. The new rule is created with the current expression type on the left-hand side and either the sequence of types for the words in the expression on the right-hand side or, in

case the expression consists of one word only, the word itself on the right-hand side; thus creating a lexicon rule.

```
foreach t in { x | x  ∈ types } do
    foreach e in { x | x ∈ t.extension } do
        r = new Rule
        r.lhs = t
        if e.length > 1 then
            r.rhs = {}
            foreach w in { x | x ∈ e.words } do
                wt = wordtypes.valueForKey(w)
                r.rhs.append(wt)
        else
            r.rhs = e.words[0]
        rules.add(r)
```

Figure 16. The Emile algorithm, stage 4b

As stated in the worst case analysis of the last stage, there are $O(sn^2)$ types and $O(sn^2)$ contexts. The inner loops are therefore reached $O(s^2n^4)$ times. The lengths of the right-hand sides are bounded by $O(n)$ and word type look-ups are $O(1)$ which results in a total running time of $O(s^2n^5)$ for stage 4.

This concludes the description of the implementation of the algorithm. As all stages are passed through once, the total running time of the algorithm is dominated by the stage with the worst running time, i.e. stage 3 with $O(s^3n^6)$. In practical experiments, however, the running time of the oracle routine is usually anything but $O(1)$, often even exponential in the length of the sentence, which means that the running time in the experiments is dominated by the oracle routine.

3.4 Experiments

3.4.1 Evaluation methods

In this section I will present some experiments with the Emile algorithm, ranging from toy problems to real world applications. In any case a grammar for the target language exists in a standard grammar formalism and is used in conjunction with an appropriate parser as oracle. The grammars learned by the Emile algorithm will be evaluated according to several criteria.

Firstly, the language defined by the Emile grammar is compared to the language defined by the original grammar. The over- and undergeneration of the Emile grammar is given in relation to size the original language, i.e. the overgeneration is $|O| / |L|$ and the undergeneration is $|U| / |L|$[10] If the original language is finite but the learned one is not, overgeneration is undefined but

10 See figure 1 on page 9.

undergeneration is still measurable. In any other case no quantitative evaluation seems possible and it will have to suffice to state symptomatic examples of over- and undergeneration.

Secondly, the quality of the resulting grammar is evaluated. In practical applications one often arrives at grammars that are highly ambiguous because certain rules that are meant to cover certain special cases are also applicable in a variety of other cases. But of course a grammar that is only as ambiguous as necessary, to cover PP-attachment ambiguity for example, is desirable and the grammars for the smaller languages have this property. The learned grammar will be evaluated by calculating the ratio of the number of the parses generated by the original grammar and by the learned grammar. Instead of calculating the individual number of parses for each sentence and then averaging over the whole corpus, it is also possible to determine the number of parse trees which can be generated by the grammar and divide it by the number of unique sentences in order to obtain a measure of the ambiguity. I will also present certain phrases that occur in the language and analyse their parses with an ad hoc interpretation.

Finally, the relation between the sentences in the sample, their complexity and sheer number, and the resulting grammar is examined. This is important as there is neither a constructive method to determine the number of examples needed and nor to asses the complexity of the sentences in the sample.

All of these evaluation methods do have certain weaknesses. In the test for undergeneration, for example, no notion of the ‘importance’ of a particular sentence is employed, i.e. each sentence that cannot be generated contributes with the same weight to the overall result, no matter how often it occurs in actual texts in the target language. Furthermore, a single ‘missing’ rule that could be responsible for the generation of a rare construction in the target language will result in a large number of sentences not being generable, especially if the missing rule would generate a symbol that in turn has a large extension. Consider the case of a missing rule for an NP-attachment of an attachment of a verb phrase. Of course, there is a large number of noun phrases in the language and therefore the lack of this rule will result in an equally large number of sentences not being generable. Because of such limitations the results presented in the next paragraphs should not be mistaken as final. They neither provide an irrefutable characterization of the algorithm nor an accurate quantitative description of it. The results obtained should only be considered indicative of certain properties and as providing an estimate of the orders of magnitude of the operations and structures involved.

3.4.2 Experiment 1: John-loves-Mary

The grammar and lexicon of this experiment describe the language that was used in section 3.2 to present the Emile algorithm. It comprises all sentences that can be formed from the two names “John” and “Mary” and the verbs “loves” and “walks.” With the customary linguistic understanding this language would be described by the following DCG rules:[11]

```
s(s(NP, VP)) --> np(NP), vp(VP).
np(np(N)) --> name(N).
vp(vp(V)) --> verbi(V).
vp(vp(V, NP)) --> verbt(V), np(NP).
```

11 See paragraph 1.2.5 on page 10. Note that the rules contain variables with which a parse tree is generated.

```
name(name(mary)) --> [mary].
name(name(john)) --> [john].
verbt(name(loves)) -->[loves].
verbi(name(walks)) --> [walks].
```

Given the example sentences “John loves Mary” and “Mary walks” the algorithm constructs a substitution matrix with 64 entries and asks 44 question. Only 6.8% of these questions are answered positively, i.e. the algorithm mainly generates sentences that are ungrammatical. It discovers 5 expression types, generates 4 rules from contexts and 8 rules for the types. These results are identical to the ones Adriaans and Knobbe state in their paper and the learnt grammar can by transformed into the one shown in paragraph 3.2.5 by renaming symbols.

The language comprises 6 sentences and none of them contains any ambiguities. With the learnt grammar, however, each of the sentences “Mary walks” and “John walks” has two parses. Either a ‘flat’ list of words as in (1) or a single symbol that contains the complete sentence.

(1) `sntc(b(mary),c(walks))`
(2) `sntc(d(b(mary),c(walks)))`

This is not problematic as parses such as number (2) could be removed easily by a post-processing stage. However, the algorithm does have a rather problematic property that can be shown when considering longer sentences such as “John loves Mary.” In this case the grammar generates five different parse trees for each such sentence.

(1) `sntc(b(john),a(loves),b(mary))`
(2) `sntc(b(john),c(a(loves),b(mary)))`
(3) `sntc(e(b(john),a(loves)),b(mary))`
(4) `sntc(d(b(john),a(loves),b(mary)))`
(5) `sntc(d(b(john),c(a(loves),b(mary))))`

When the symbols are removed from the parse trees it becomes obvious that each possible bracketing for a sequence of three words is generated.

(1) [John loves Mary]
(2) [John [loves Mary]]
(3) [[John loves] Mary]

Considering the fact that the underlying model, the Lambek calculus, is structurally complete[12] this effect is probably not surprising. Nonetheless, it is undesirable in practical applications. Parse trees (4) and (5) are the result of the same rule that caused parse tree (2) in the two-word sentences. It is interesting to note that a sixth parse tree is ‘missing.’

(6) `sntc(d(e(b(john),a(loves)),b(mary)))`

12 See paragraph 1.4.3 on page 15.

This illustrates that the learnt grammar while being affected by the structural completeness Lambek calculus does not necessarily generate all possible bracketings when one bracketing can be generated.

The ambiguity for the sentences with “walks” is 2 and there are 2 different sentences of this kind. The ambiguity for 4 sentences with “loves” is 5. Hence, the average ambiguity of a sentence given the learnt grammar is 4.

3.4.3 Experiment 2: A, B and C

This experiment is from Adriaans’ and Knobbe’s paper as well and I repeat it here to show that my implementation based on the interpretation of their paper does reproduce their results. The language comprises all words consisting of a number of “ab” followed by a final “c” and it can be described with the following context-free grammar:

(1) $S \rightarrow A$ c
(2) $A \rightarrow$ a b | A a b

The sample consists of the words “abc” and “ababc” providing the two most simple examples possible.

The algorithm generates 21 basic rules with 15 contexts and 12 expressions from this sample, resulting in a substitution matrix with 192 entries. Of these entries 93 form duplicate sentences which means that the oracle has to be queried 99 times. The algorithm constructs a grammar consisting of 23 rules[13] which do exactly generate the original language. As in the previous experiment, however, the rule set is redundant and the language could be generated with a small subset of the learnt rules such as the following one:

(1) $S \rightarrow G$
(2) $G \rightarrow C\ B\ A$
(3) $C \rightarrow$ a | $C\ B\ C$
(4) $B \rightarrow$ b | $B\ C\ B$
(5) $A \rightarrow$ c

The subset contains a type C which generates lists of “ab” followed by an “a” and a type B that generates lists of “ba” followed by a “b,” but a type that generates lists of “ab” only cannot be found in the rule set. This is interesting to note as it shows that the grammar, while being correct, is not ‘optimal.’ Because there are various ways to generate sequences of a and b and there are rules which allow for changing from one strategy to another in the middle of a word, the number of parse trees is exponential in the length of the sentence. The original grammar is unabiguous and imposes no bound on the length of its words. Hence, with l denoting the length of a sentence, the average ambiguity can be approximated by $\lim_{l \rightarrow \infty} e^l / l$ and is therefore infinite.

All the results match exactly those presented in Adriaans’ and Knobbe’s paper. It therefore seems likely that my implementation does in fact represent the Emile algorithm but it is obviously neither a prove nor does it rule out that subtle differences exist.

13 See table 4 in appendix A.

3.4.4 Experiment 3: John-loves-Mary extended

For this experiment the language of the first experiment was extended by a few words and constructs which are suitable to illustrate some phenomena occurring in the English language. It is, however, at any rate still a toy problem. Noun phrases were extended such that they are constituted not only by proper nouns but also by generic nouns such as “girl” and “telescope” together with a determiner, “a” and “the” in this case. The number of verbs was increased and the grammar was augmented to account for so-called θ-roles. This is a verb-driven approach to sentence structure in which each verb has by definition a ‘thematic grid.’ This grid contains a number of slots describing which arguments a verb takes and of what type these arguments must be. All slots are mandatory but a verb may have more than one grid thus allowing for a varying number of arguments. The arguments include all participants in the ‘action’ described by the verb, also including the subject even though this is not dominated by the verb in the parse tree. The type of the argument is given as a θ-role from a small set of roles such as ACTOR, THEME, GOAL, BENEFACTOR etc. Note that there is no general agreement about the number and meaning of these θ-roles. In the example, θ-roles are limited to AGENT, ITEM and INSTR, the latter comprising only instruments that can be used for seeing. The following figure shows thematic grids for the verbs “walk” and “see.”

walks
NP AGENT

sees		
NP AGENT	NP (ANY)	PP INSTR

Due to the θ-roles a fine grained distinction between various noun phrases is made: “a man” is an AGENT and can thus occur as a noun phrase before a verb while “a flower” cannot. Noun phrases can also have adjuncts but these can only be instruments and items which means that “John with the flower” is a legal noun phrase while “John with Mary” is not. Furthermore, the verb “sees,” having among others the thematic grid shown above, allows for sentences with a PP-attachment ambiguity as illustrated by the following bracketings:

(1) [John] sees [Mary [with [the telescope]]]

(2) [John] sees [Mary] [with [the telescope]]

Because the language contains words such as “flower” which is an item but not an instrument, and can therefore be an adjunct to an NP but not to the VP, a learning algorithm has the possibility to discover this ambiguity. When it replaces “the telescope” with “the flower” in the sentence above the oracle will accept the sentence because it can parse it in a way analogous to the one depicted in (1).

(3) [John] sees [Mary [with [the flower]]]

However, when “Mary” is replaced by an NP with adjunct the algorithm can construct sentences that show the difference between the two structures.

(4) [John] sees [Mary [with [the flower]]] [with [the telescope]]

(5) *[John] sees [Mary [with [the flower]]] [with [the flower]]

The language can be described with a grammar in the Gulp formalism with 13 rules and 19 lexicon entries[14] and comprises 7830 different parse trees describing 7470 different sentences. The average ambiguity of each sentence in the original language is therefore 1.05.

As the measure for the complexity of a sentence is not constructive in Adriaans' work, an estimate had to be found that could be calculated from the sentence and its structural description. I chose to remove the word symbols from the generated grammar and then used the number of opening brackets in the structural description of each sentence as its degree of complexity. The following example shows a few parses and the estimated complexity of the sentence:

s(np(Mary), vp(walks)) – 3
s(np(The, girl), vp(walks)) – 3
s(np(John), vp(loves, np(Mary)) – 4
s(np(np(The, girl), pp(with, np(the, telescope))), vp(walks, pp(with, np(the, man)))) – 8

If the word symbols had not been removed then each word would have been bracketed individually and the degree of complexity would have been increased by the number of words in each sentence. Also, each occurrence of a compound NP such as "the girl" would have increased the complexity by two. I felt that such a measure of complexity would not be as adequate as too much emphasis would have been placed on the sheer number of words in the sentence rather than the structural complexity.

According to the measure presented above the simplest sentences of the language have a complexity of 3 and the most complex ones have a complexity of 12. Note that due to permutation of phrases and words in phrases the number of sentences grows exponentially for each level of complexity.

The probability for a sentence returned by the examples routine should decrease exponentially with its complexity. Hence, the probability distribution is $P(c) = e^{-(1/n)c}$ where c is the complexity and n a small constant. The probabilities for all $c \in [3, 12]$ are multiplied with a constant such that their sum equals the desired sample size. The actual number of sentences for each level of complexity can then be determined by rounding this value to the nearest integer. The left half of figure 17 shows the result for $n \in \{2, 4, 6\}$ for a sample S with 20 sentences. Note that the language contains only 6 sentences of complexity 3 which means that small n cannot be used properly, i.e. not enough examples of the required complexity can be provided. Large n on the other hand diminish the effect of the exponential distribution and therefore I decided on $n = 4$. The right half of figure 17 shows the number of sentences required for several sample sizes. For each sample the sentences of each level of complexity were drawn at random. I believe that this procedure approximates the required properties of the examples routine

During the course of this experiment more than 30 different samples with varying sizes were tested and the results for 6 runs that are somewhat representative are summarized in figure 18. Several points are noteworthy. It is obvious that the random selection of sentences of a certain degree of complexity can affect the size of the substitution matrix considerably. However, this is generally only true for smaller samples and is simply the result of certain constructions of the language not being present in the sample. Also, the prediction that the size of the substitution matrix will be considerably smaller than the worst case of $O(s^2n^4)$ is supported by the experiment and it can be seen that with increasing sample size the relative size of the matrix decreases as more and more repetitions are present in the sample. Interestingly, the proportion between the size of the matrix and the number of questions seems to be rather constant.

14 See appendix B.

c	$n = 2$	$n = 4$	$n = 6$	$n = 4$		
	$\|S\| = 20$			$\|S\| = 20$	$\|S\| = 30$	$\|S\| = 40$
3	8	5	4	5	7	10
4	5	4	3	4	6	8
5	3	3	3	3	4	6
6	2	2	2	2	3	4
7	1	2	2	2	3	3
8	1	1	2	1	2	3
9	0	1	1	1	2	2
10	0	1	1	1	1	2
11	0	1	1	1	1	1
12	0	0	1	0	1	1

Figure 17. Number of sentences by sample size and n.

sample number	1	2	3	4	5	6
sample size (*s*)	20	20	30	30	40	40
longest sentence (*n*)	12	13	13	13	13	13
contexts	395	504	586	599	765	759
expressions	288	365	410	437	496	530
matrix size	113 760	183 960	240 260	261 763	379 449	420 270
...in percent of s^2n^4	1.37%	1.61%	0.93%	1.02%	0.83%	0.88%
questions	88 372	143 079	188 957	205 734	295.714	318 043
...in percent of matrix	77.7%	77.8%	78.6%	78.6%	77.9%	79.1%
positive answers	0.8%	0.7%	0.7%	0.7%	0.5%	0.6%
types	118	131	132	137	135	137
rules	800	1 165	1 227	1 332	1 398	1 495
...from contexts	579	872	904	990	1 032	1 089
undergeneration	0.362	0.058	0.045	0.019	0.169	0
overgeneration	0	0	0	0	0	0
avg. ambiguity	13.31	20.39	24.56	27.31	28.13	31.17

Figure 18. Results from 6 runs.

The number of types increases as more possible constructions are found with the help of larger substitution matrices. Their number does however seem to converge against constant which consequently must be considered sufficient to represent the target language. For the example this value appears to be somewhere near 140. It is also plausible that with a larger substitution matrix, and thus more types, more complete grammars are learnt. This is generally true, but as sample 5 illustrates it is not necessarily the case because despite the comparatively large sample size it is still possible that due to the random nature of the selection of example sentences constructions can be missing in the sample. It is only the probability for such a case that does decrease with larger sample sizes.

```
sntc(john, bd(meets, the), man, with, the, telescope)
sntc(bb(john, meets), the, man, with, the, telescope)
sntc(bb(john, meets), h(the, man), with, the, telescope)
sntc(john, w(meets, h(the, man)), with, the, telescope)
sntc(john, w(meets, the, man), with, the, telescope)
sntc(john, meets, the, man, with, the, telescope)
sntc(bm(john, meets, the), man, with, the, telescope)
sntc(john, bt(meets, h(the, man), with), the, telescope)
sntc(john, meets, the, ch(man, with, the, telescope))
sntc(john, ci(meets, h(the, man), with, the), telescope)
sntc(cl(john, meets, the, man), with, the, telescope)
sntc(john, meets, ct(the, man, with, the, telescope))
sntc(john, meets, ct(h(the, man), with, the, telescope))
```

Figure 19. Parse trees for "John meets the man with the telescope."

The number of sentences needed to learn a complete, i.e. weakly equivalent grammar, for the target language with a reasonably high probability seems to be approximately 40. Sample #2 was 'tuned' manually, i.e. the sentences were not selected according to the examples routine but such that they would exemplify all constructs of the target language, in order to investigate whether it would be possible, even though with a very low probability, to learn a complete grammar with a significantly smaller sample.

Regarding the number of rules, no bound similar to the one for the types seems to exist, or it is too large to become evident with samples comprising less than 50 sentences. This does inevitably result in a constantly increasing ambiguity of the grammar. Figure 19 shows all parse trees generated by the grammar learnt from sample #6 for a particular sentence.

Regarding the quality of the rules, it can be said that the algorithm has identified certain types that correspond to the customary linguistic understanding such as *H* and *CT*, representing simple noun phrases and noun phrases with adjunct, and *W* which comprises verb phrases with simple nouns as objects. It is interesting to note that no general type for a verb phrases was found. Furthermore, the lack of a type for PPs is tolerable but the existence of types such as *BD* and *CI* that cut through sequences of words that should be considered units in any grammar formalism is less acceptable. It is likely that these rules stem from the tendency of the algorithm to construct rules for each possible bracketing of a valid sentence as described before.

3.4.5 Experiment 4: The 'Call' database

This experiment is based on real-world data and is an evaluation for one of the areas in which a practical application of the Emile algorithm was envisioned. The basis for this experiment is a database with descriptions of service incidents reported by the operators of the hot-line of a large computer manufacturer. After answering a phone call the operators write down a brief, i.e. one or two sentence description of the problem and the suggested solution. If known, they also state whether the problem could be solved.The language used in this database consists of a subset of English limited to function words and technical vocabulary with few adjectives and almost no ambiguous use of words, but it also contains a large number of abbreviations and omissions which are not normally found in written English. Furthermore, the sentences are usu-

ally fairly long and often contain multiple subordinate clauses. For the manufacturer it would be interesting to apply data-mining techniques to this database. These require a structured input and it was hoped that the Emile algorithm could learn a grammar for the target language and parse the sentences in the database. It does not seem necessary that the structure is linguistically relevant as long as it is consistent.

Given the results from the last experiments it is obvious that a larger number of examples is necessary to achieve any meaningful results. However, even moderate samples with less than 100 sentences result in substitution matrices with well over 10^6 entries. For this thesis it was clearly impossible to answer such an amount of questions. For an application at the manufacturer it would be a matter of weeks and it seems more than questionable whether a team or even a single persons could answer all questions consistently; not even considering that the task of giving consistent answered is further complicated by the fact that the language used in the Call database is not standard English and therefore the intuitive judgement of a sentence would often be inconsistent with the target language.

At the same time a project aimed at developing a hand-crafted grammar for this corpus was pursued.[15] This grammar is expressed in the PATR formalism[16]and suffers from the same problems that can be found in many such systems. The grammar is highly ambiguous because rules added for certain special cases also apply in many other cases, an extension of the grammar to cater for an uncovered examples renders other parts of the grammars useless and, especially interesting in the context of this application, an inefficient parser.

As it was impossible to answer the oracle questions manually it seemed interesting to use the hand-crafted grammar as an oracle. While this might at first sight appear to be absurd for a practical application it does make sense if one considers the shortcomings of the grammar. An engineer could use the Emile algorithm as a tool in the process of creating a ‘good’ grammar. It would be possible to write a version of the grammar in an expressive formalism such as PATR and then transform this into a less redundant, context-free grammar using the Emile algorithm. While the former seems not very realistic given the results of the previous experiments, the latter is a valid point as the grammars generated by the Emile algorithm are not only context-free by also conform to the restrictions for Tomita’s algorithm[17] and thus parsing could be performed in linear instead of exponential time. It remains to be seen though, whether the alterations in the ruleset are tolerable for the application.

To perform the experiments the Emile algorithm was connected to the PC-PATR program[18] via UNIX pipes and the for each question the Emile algorithm ‘typed in’ a sentence. Depending on whether the output indicated at least one parse the sentences was considered correct or not. Various strategies for determining a sample were tried but all attempts to learn any meaningful grammar from sentences in the database failed. This is most likely due to the comparatively limited sizes of the samples. Adriaans estimates that approximately 5 million sentences sampled according to **m** are needed to learn a language with 50 000 words[19] but on any contempo-

15 [Hölscher, 97]

16 See paragraph 1.2.5 on page 10.

17 *ibid.*

18 The software can be obtained from the web site of the Summer Institute of Linguistics at <http://www.sil.org> or by contacting its headquarters, the International Linguistics Center in Dallas, TX.

rary workstation sample sizes in the range of 500 sentences are on the borderline of tractability. The rate at which the PC-PATR program parses sentences given the existing grammar was found to be roughly $m / 15$ sentences per second where m is the MIPS value of the machine. This means that a workstation with 150 MIPS can parse 10 sentences per second which means in turn that the exploration of a substitution matrix with 10^8 entries which can easily result from a sample with 500 sentences would take well over 100 days. Another problem with the database is that the majority of the sentences are too long, i.e. the samples cannot not be taken according to a distribution in which shorter sentences occur with an exponentially higher probability than longer, more complex sentences. Most sentences contain a large variety of grammatical constructions and it would therefore be necessary that a trained person writes a sample containing short sentences each illustrating few or even only one construction. This, however, seems at least as complicated as directly writing a grammar.

The experiences with the sentences in the Call database show that it seems to be too ambitious to learn a grammar for a larger subset of English with the current implementation of the Emile algorithm. Furthermore, any improvements that reduce the size of the substitution matrix would have to reduce it by orders of magnitude to provide any improvement.

3.5 Conclusions

3.5.1 Learnability

Even given the ad hoc nature of the example selection, the experiments do support Adriaans' learnability results. If enough example sentences are provided the algorithm constructs in polynomial time a grammar that is weakly equivalent to the original grammar. However, it has also become obvious that at least two problems have to be overcome before a practical application of the Emile algorithm is possible. I will discuss these problems in the following paragraphs.

3.5.2 The strained oracle

The experiments substantiate the initial suspicion that the size of the hypotheses space, while being considerably smaller than the one for an exhaustive search, is still by far too large to be checked with the help of a competent speaker of the target language. I say competent because for an untrained person it seems to be difficult or even impossible to judge whether a sentence is syntactically correct while completely disregarding its meaning. In fact, it is a long standing debate what level of correctness the notion of syntactical correctness entails. Chomsky is often quoted to have said that he believes the sentence "Colorless green ideas sleep furiously" is syntactically correct but it is also obvious that most persons without a background in linguistics would reject such a sentence as 'wrong.' To make matters worse, not only meaning but also performance limitations impede a systematic behaviour of a speaker as an oracle. Consider the following example:[20]

19 [Adriaans, 92], p. 132.

20 From [Haegeman, 91], pp. 7–8

(1) Bill had left. It was clear.
(2) [That Bill had left] was clear.
(3) It was clear [that Bill had left].
(4) Once that [it was clear [that Bill had left]], we gave up.
(5) Once that [[that Bill had left] was clear], we gave up.

Two independent sentences are presented in (1). Sentences (2) and (3) illustrate two possibilities how one of the sentences can be made a subordinate clause of the other. In the construction of sentence (4) a similar principle is applied and sentence (3) is made a subordinate clause of a more complex sentence. However, the application of the same principle to make sentence (2) a subordinate clause as shown in (5) results in a sentence which is clearly unacceptable for most speakers of English even though it is grammatical in a strict sense. If the oracle rejects sentence (5) then the systematic rules in the grammar that govern subordination will have to be extended to account for performance limitations of human speakers. A result that is not really desirable.

On the other hand the interaction with the oracle is essential for the algorithm. As the size of the hypotheses space was reduced considerably by introducing knowledge about the domain it seems reasonable to suspect that its size can be reduced further by incorporating more domain knowledge. Also, the size of the hypotheses space grows polynomially with the number of sentences in the sample. It would therefore seem plausible to divide the examples necessary to learn the language into small batches and let the algorithm run anew for each batch. It is not clear, however, how the knowledge acquired in one pass can be transferred into the next one. Chapter 5 deals with these topics

3.5.3 Redundant rule set

The second main obstacle for a practical application of the Emile algorithm is its tendency to create extremely redundant rule sets. This is problematic insofar as the Emile algorithm was intended to be used as a preprocessing stage that would provide structured representations for another algorithm. For obvious reasons it is hard to find regularities in the structural description of a set of sentences if for each sentence itself a variety of descriptions exists. Another consequence will emerge in the discussion of an incremental version of the Emile algorithm in section 5.3.3.

This property of the Emile algorithm is partially due to the structural completeness of the Lambek calculus and is as such an intrinsic property of the algorithm. However, there are certain standard procedures to manipulate sets of rules or clauses and in the following chapter I will explore if and to what extend they can alleviate the problem.

Chapter

4 Improving the rule set

As explained before, the redundancy of the set of rules is an intrinsic property of the algorithm and therefore countermeasures are limited to curing the symptoms rather than eliminating the cause. In sections 4.1 and 4.2 I describe two applications of standard procedures with which some redundant rules can be eliminated and the quality according to some linguistic intuition can be improved. In section 4.3 I return to the problem of homonymous and polysemous words and reconsider the current implementation.

4.1 Using depth-limited unfolding to discover redundancies

4.1.1 Theory restructuring

The field of theory restructuring has been studied for various domains. What is common to all is the goal to restructure a set of rules stated in a fixed formalism such that it is improved according to a given criterion while retaining the extension. In terms of grammar theory the resulting grammar has to be weakly equivalent to the original one. However, this limitation is not strict enough for our purposes because it is assumed that the grammar learned by the Emile algorithm has some linguistic relevance and excessive modifications of the set of rules would shift the focus from the Emile algorithm to the restructuring algorithm. In an extreme model the Emile algorithm would provide nothing more than a description of the extension of the language to the restructuring algorithm and it would be left to the latter to discover the structure in it. The Emile algorithm would be reduced to an examples routine in a meta-algorithm, a concept which is clearly not the subject matter of this thesis. I will therefore only consider modifications that do not substantially change the structure of the rule set.

Most of the applications of theory restructuring are located in the area of knowledge bases and therefore the formalisms involved are usually propositional logic or FOPC. While CFGs can obviously be treated as a special case of FOPC and therefore results from these applications could be transferred to CFGs this is generally not practical because the majority of the procedures are based on regularities in the variables of the predicates[1] but CFGs rewritten in FOPC consist of predicates without variables. There are, however, certain general principles that can be applied successfully to the problem domain of the Emile algorithm.

Folding and unfolding describe two operations on the set of rules that are similar to the activities of a parser that uses these rules to analyse some input. Unfolding describes the expansion of the left-hand side of a rule into its right-hand side somewhere in another rule. Folding is the inverse operation and thus describes the replacement of the symbols of a right-hand side of some rule *r* that occur somewhere on the right-hand side of a longer rule by the left-hand side of *r*. Consider the following example.

1 For example ‘Fender’ in [Sommer, 96].

$\boldsymbol{R}$ = {$A \rightarrow X\ C, B \rightarrow D\ X\ E, X \rightarrow Y\ Z$}
$\boldsymbol{R_u}$ = {$A \rightarrow Y\ Z\ C, B \rightarrow D\ Y\ Z\ E, X \rightarrow Y\ Z$}

The set $\boldsymbol{R}$ is transformed into $\boldsymbol{R_u}$ by unfolding X. In this case folding with the rule for X in $\boldsymbol{R_u}$ would indeed be the exact inverse operation and result in set $\boldsymbol{R}$. This is not necessarily the case as the order in which the folding is performed as well as the precedence given in cases where more than one rule could be folded influences the result.

4.1.2 Description

When analysing the results of the experiments I stated that the majority of rules are created from the contexts. And because the contexts are the result of a systematic division of the input sentences they bear a lot of similarity. Consider the example sentence “a b c” and the context/ expression pairs created from it.

(1) S/bc → a
(2) a\S/c → b
(3) ab/S → c
(4) a\S → bc
(5) S/c → ab
(6) S → abc

The algorithm might create the following rules (among others):

(1) S → $A\ B\ C$
(2) S → $X\ B\ C$
(3) S → $Y\ C$
(4) S → Z
(5) $A \rightarrow a, B \rightarrow b, C \rightarrow c$
(6) $X \rightarrow A \mid C$
(7) $Y \rightarrow A\ B$
(8) Z → $Y\ C$

These rules reflect to some degree the structure of the substitution matrix. Rules (3) and (7) probably stem from the insertion of the expression “ab” into the original context and the same might be the case for rule (2), the first part of (6) and the expression “a.” As these rules are very similar it is often enough to unfold a single symbol to reach equivalency with another rule. For example, unfolding Y in rule (3) immediately yields rule (1), meaning that one of the rules can be removed without changing the language defined by the set of rules.

The procedure for an improvement of the set of rules is the following: In all sets of rules with the same symbol on the left-hand side the algorithm successively fixes each rule and unfolds symbols in the remaining rules of the subset. If an equivalence is found the rule is discarded. Lexicon rules are ignored by the algorithm as unfolding them would result in terminal symbols, i.e. words of the language, appearing in the rules. And these appearances were explicitly removed in the generalisation stage. In order to guarantee a polynomial running time the algorithm does not unfold symbols that are the product of an unfold operation, i.e. it limits the search depth to 1. Given this restriction another problem deserves consideration. In the example it might happen that the algorithm discards rule (3) on the basis of rule (4) and (8) before

checking whether rule (1) is redundant. A path is left from *Z* to *ABC* via rules (8) and (7) but that would involve two levels of unfolding and the redundancy would therefore remain undiscovered. Fortunately, there is a simple modification to avoid this problem. Instead of processing the rules in each subset in random order the algorithm processes the rules in order of their lengths, starting with the longest one. The same problem occurs on a larger scale as well when certain rule sets are modified before others. Because of that the algorithm retains copies of the original sets and uses these in the unfolding operations.

Regarding the example the algorithm examines rule (1) first, discovers that it is redundant on the basis of rules (2) and (6) and removes it. Rule (2) is kept because no unfolding of any rule of type *S* can produce its right-hand side. The next rule to check is number (3) and it is removed because its redundancy can be shown with rules (4) and (8). Now no subset containing more than one rule is left and the algorithm halts. It is clear that the resulting set of rules still contains redundancies as there are two derivations for the sentence “a b c” for example:

- $S \rightarrow XBC \rightarrow ABC \rightarrow aBC \rightarrow abC \rightarrow abc$
- $S \rightarrow Z \rightarrow YC \rightarrow ABC \rightarrow aBC \rightarrow abC \rightarrow abc$

The problem is that from a longer rule, namely rule (2), a superset of a shorter rule, number (4) in this case, can be derived. However, it seems that from a linguistic point of view a stratified rule set is more desirable and therefore this behaviour is not a disadvantage. This means that instead of finding ways to remove the shorter rule one should seek to place restrictions on the longer rule such that it generates only those sequences of symbols which cannot be generated by the shorter rule. In the example this could be achieved by changing rule (2) to

(6b) $S \rightarrow C\ B\ C$

Furthermore, with the algorithm described above there is still one pathological case that has to be considered:

(1) $S \rightarrow Y\ B\ C$
(2) $S \rightarrow X\ B\ C$
(3) $X \rightarrow Y$
(4) $Y \rightarrow X$

Given this set of rules the algorithm removes rule (1) on the basis of rules (2) and (3). If it used a copy of all rule sets as outlined above then it would subsequently remove rule (2) on the basis of rules (1) and (4) which is obviously wrong. It is arguable whether such a case can occur but as it is possible to guard the algorithm against this problem this consideration is not necessary. The solution is to check for each rule of the same length as the current rule whether it is still present in the set of all rules and skip the redundancy test if this is not the case. In the example the algorithm cannot show the redundancy of rule (2) because rule (1) is not available any longer.

4.1.3 An efficient implementation

The implementation of the improvement described above needs a dictionary that contains the types as keys and sets with all rules of that type as values. This dictionary is created as a by-product in stage 4 in the implementation and it is named *rulesByType*.

```
foreach t in { x | x ∈ types } do
    ruleset = originalRulesByType.valueForKey(t)
    foreach cr in { x | x ∈ ruleset } inReverseOrder do
        foreach r in { x | x ∈ ruleset\{cr} } do
            if r.length > cr.length
                continue with next cr
            if not rules.contains(r) then
                continue with next r
            if match(cr.rhs, r.rhs) then
                rulesByTypes.valueForKey(t).remove(cr)
                rules.remove(cr)
                continue with next cr
```

Figure 20. Outer loop for discovering redundancies by unfolding

First of all the algorithm creates a copy of the *rulesByTypes* dictionary named *originalRulesByType* and in the copy it changes the data structure of the values from a set to an array in which the rules are sorted according to their length. This is trivial and not shown in figure 20.

The algorithm then iterates over all types and for each type successively fixes each rule, starting with the longest one. It then checks for this current rule whether its redundancy can be shown by unfolding symbols in other rules of the same type. To do so, it iterates over all rules of the same type which are shorter or as long as the current rule and calls a subroutine *match* that performs the actual test. If both rules can be shown to be equivalent to the current rule it removes the current rule and continues with the next rule in the same set. Note that the tests for lexicon rules are omitted in figure 20.

In the worst case scenario from paragraph 3.3.6 each expression is valid in all but one context and at the same time all expressions have a different type. This results in $O(s^2n^4)$ rules of the sentence type. The two inner loops iterate over all rules of the current type and therefore for the sentence type the body can be reached $O(s^4n^8)$ times. All other types consist of singleton rules as each expression has a different type and the inner loop is not reached at all.

The match routine takes as arguments two sequences of symbols *a1* and *a2* which are to be compared and it returns ‘true’ if both sequences can be matched by unfolding symbols in *a2* and ‘false’ otherwise. The routine first skips over a possible common prefix of both sequences. If it reaches the end of both, the match is successful and the routine returns ‘true.’ If it reaches the end of one of the sequences only, then no match is possible and ‘false’ is returned. In all other cases there is a least one symbol left in both sequences. The routine retrieves the set of rules with which the current symbol in *a2* can be unfolded and iterates over the rules of this set. For each rule it first checks whether the length of the sequence resulting from replacing the current symbol in *a2* by the right-hand side of the rule is longer than *a1* and if this is the case skips the unfolding because no complete match would be possible. The length of the common prefix is not taken into account as it is the same for *a1* and *a2*. The routine then checks whether the symbols on the right-hand side match those of *a1* starting at the position after the common prefix. If this is the case it calls itself recursively with the rest of *a1* and the rest of *a2*.

```
i = 0
while (i < min(a1.length, a2.length)) and (a1[i] == a2[i]) do
    i = i + 1
if (i == a1.length) and (i == a2.length) then
    return true
if (i == a1.length) or (i == a2.length) then
    return false
unfoldpos = i
t = a2[unfoldpos]
ruleset = originalRulesByType.valueForKey(t)
foreach r in { x | x ∈ ruleset } do
    if(a2.length + r.rhs.length - 1 > a1.length)
        continue with next r
    if rhs.isSubarrayOfArrayAtIndex(a1, unfoldpos) then
        rest1 = a1.subarray([i, a1.length - i])
        rest2 = a2.subarray([unfoldpos+1, a2.length-unfoldpos-1])
        if match(rest1, rest2) then
            return true
return false
```

Figure 21. The match routine

A worst case analysis of this routine, especially in conjunction with the outer loops from which it is invoked, is extremely complicated. I will therefore only analyze the worst case of the match routine itself. If the routine can be shown to run in time polynomially bounded by the input length then the complete algorithm runs in polynomially bounded time because the number of invocations and the length of the arguments of the routine are polynomially bounded.

The routine tries to match a longer rule by unfolding symbols with the help of auxilliary rules in all shorter rules of the same type as the longer rule. For each rule pair there is a number of possible combinations of unfoldings in the shorter rule and because the number of combinations grows exponentially in the length of the sequence one could assume that this would result in an exponential running time. However, the algorithm only tries those combinations for which a rule exists and is completely matched by the next part in the longer rule. The numbers of right-hand sides of all auxilliary rules that can match any part of the right-hand side of the longer rule is clearly bounded by the length of the right-hand side of the longer rule, and thus bounded polynomially in the length of the input. Each such right-hand side can only be combined with any of the symbols in the right-hand side of the shorter rule, the number of which is polynomially bounded as well. Hence, the possible number of combinations that can actually occur with the rules, is the product of two polynomially bounded quantities and thus polynomially bounded itself.

4.1.4 Experiments

In experiments it was found that the running time of the algorithm was comparatively short, thus supporting the intuition of an average-case running time with a polynomial bound of a small order. The number of rules is reduced considerably but mainly the rules generated from

contexts are affected as can be seen in figure 22. While this is obviously a positive result it also demonstrates that the structural redundancy of the rule set cannot be removed as easily. Along with the reduction of the number of the rules, the average ambiguity drops. It remains reasonably high, though.

Experiment / Sample	2		3 / #2		3 / #6	
Algorithm	orig.	impr.	orig.	impr.	orig.	impr.
rules	23	21	1 165	807	1 495	1 003
...from contexts	11	9	872	560	1 089	677
...from expressions	12	12	293	247	406	326
average ambiguity	∞	∞	20.4	12.9	31.2	18.7

Figure 22. Results of using unfolding to remove redundant rules.

Regarding the analysis of the parses of the sentence “John loves Mary” on page 51 the improvement presented in this section would successfully remove parses (4) and (5) but would generally not affect sets of rules such as (1) – (3).

4.2 Folding singleton rules

4.2.1 Description

In the generalisation stage each word in an expression or a context is replaced by its type on an individual basis. For long sequences of words this obviously results in long sequences of symbols and thus in long, ‘flat’ rules. At the same time long sequences of symbols are also expressions and belong to an expression type. Recalling that expression types are translated into rules with the type on the left-hand side and the sequences of symbols constituting the expression on the right-hand side, this implies that if some expression is contained in the right-hand side of a rule then this rule can be shortened by replacing the symbols of the expression with the type of the expression, i.e. by a folding operation. Consider the following example:

(1) $P \rightarrow A\ X\ Y\ B$

(2) $Q \rightarrow X\ Y$

(1b) $P \rightarrow A\ Q\ B$

The symbols *XY* in rule (1) can be folded with rule (2) into *Q* creating rule (1b). If the latter had existed before, a redundancy would have been shown and one of the rules could be removed. This, however, describes the improvement presented in the last section. In this section I will consider the case in which rule (1b) does not exist and ask whether it would be legal to remove rule (1) in favour of rule (1b). In the example this question can be answered positively as the set of symbols that can be derived from *P* remains the same. That this is not generally the case though can be illustrated by adding the following rule to the example set:

(3) $Q \rightarrow Z$

With rule (1) the extension of *P* remains {*AXYB*} while the extension of *P* as defined in rule (1b) changes to {*AXYB, AZB*}. This is not acceptable as allowing overgeneration on this scale

would result in an even more redundant rule set. Note that if rule (4) is added the folding becomes legal again.

(4) $P \rightarrow A\ Z\ B$

Now, a clear pattern emerges: Folding a sequence of symbols, i.e. an expression, into its type is permissible if and only if all other expressions of the same type can occur at the same position in the original rule. The problem with this statement is that the test whether all other expressions can occur in the original rule can be arbitrarily expensive. In the example an algorithm might find the right-hand side of rule (2) in rule (1). Before replacing *XZ* with *Q* it would have to check whether replacing *XY* with all other right-hand sides of *Q* would result in a right-hand side of *P.* In this case the test is easy as the only other right-hand side of *Q* is *Z* and the corresponding sequence *AZB* is the right-hand side of another rule for *P.* However, if rule (4) is replaced with rule (4b) and either rule (5) or (6) the folding would still be valid but it would need another unfolding to discover that fact.

(4b) $P \rightarrow A\ D$
(5) $D \rightarrow$ Z B
(6) $Z \rightarrow A\ D$

It is intuitively clear that a corresponding algorithm would have a highly exponential running time because the complete extension for most symbols would have to be generated. As in the previous section the introduction of a depth-limit would help but preliminary experiments have shown that another limitation is preferable: The set of rules contains a large number of singleton rules, i.e. rules that stem from types which consist of one expression only. If a right-hand side of a singleton rule is found anywhere it can be readily replaced by the symbol on the left-hand side as there are simply no other right-hand sides which have to be checked.

The algorithm proposed in this section iterates over all rules and recursively applies as many foldings as possible using singleton rules. The order in which right-hand sides of singletons are searched in the longer rule is a crude heuristic. In English and many other languages the head of a phrase, the noun in an NP for example, is usually located at the left and adjuncts are usually added right of the head. Therefore, the algorithm scans the rule from the right to the left and performs the first folding possible. It then starts over at the very right again. If it discovers a redundancy the corresponding rule is removed, otherwise the rule is replaced by the shortest rule resulting from the foldings. As there is no depth limit, the considerations stated in the last section regarding the sequence in which the redundancy checks have to be performed do not apply. However, in this case there is a certain sequence which is preferable for efficiency reasons. Folding operations in a rule can only be performed with shorter rules and it is obviously an advantage if the number of shorter rules is as small as possible. If shorter rules are checked first, then it is assured that the algorithm does not perform any unnecessary foldings in longer rules.

4.2.2 An implementation

The algorithm first creates a sorted representation of the set of rules. It then iterates over all rules starting with the shortest ones, skipping those with less than two symbols on the right-hand side. For each rule it collects in the array *shorterSingletonRules* all singleton rules which are shorter than the current rule, i.e. all rules that can possibly be used in folding operations. The actual shortening is done in the subroutine *shortenRule* which always holds in its second argument the shortest possible version of the current rule. If this is *nil* then the current rule was

```
do
    numberOfShortenings = 0
    sortedRules = rules.sortByLength()
    foreach cr in { x | x ∈ sortedRules } do
        if cr.length < 3 then
            continue with next cr
        rulesOfSameType = rulesByType.valueForKey(cr.lhs)
        if rulesOfSameType.count() == 1 then
            continue with next cr
        if cr.length ≤ rulesOfSameType.shortestRule().length then
            continue with next cr
        shorterSingletonRules = new Array
        foreach r in { x | x ∈ rules } do
            if r.length ≤ cr.length then
                if rulesByType.valueForKey(c.lhs).count == 1 then
                    shorterSingletonRules.add(r)
        bestProposal = cr
        shortenRule(cr, (byref)bestProposal)
        if bestProposal != cr then
            rules.remove(cr)
            rulesOfSameType.remove(cr)
            if bestProposal != nil then
                rules.add(bestProposal)
                rulesOfSameType.remove(bestProposal)
                numberOfShortenings += 1
while numberOfShortenings > 0
```

Figure 23. Outer loop for shortening rules by folding singletons

found to be redundant and can be removed. If a shorter rule was found, i.e. the rule returned from the subroutine is not the original rule, then the original rule is removed and the shorter rule is added to the relevant sets. Because a shortening which was introduced might allow for a shortening of a rule that was checked before, the whole procedure is repeated until no further shortenings are found.

As stated before the shortening routine takes two arguments: A rule *cr* which is to be shortened by folding operations and, by reference, the result from other operations giving the shortest possible replacement so far, named *best*. Only if the routine finds a yet shorter rule it may store its result in this argument. The routine iterates over all possible indices for a folding and for each index over all candidate rules, skipping those which are too long to be folded at the current position. If it finds the right-hand side of the candidate rule in the current rule then the folding is carried out. If the result is a member of the set of all rules a redundancy has been discovered and the routine terminates accordingly. Otherwise it checks whether the new rule represents the best attempt so far and sets *best* accordingly. In then tries to shorten the new rule even further and returns immediately if a redundancy is discovered. In all other cases the next rule and/or index is tried.

```
for j = cr.length - 1 downto 0 do
    foreach r in { x | x ∈ shorterSingletonRules } do
        if (r.length + j ≤ cr.length) and (r != cr) then
            if r.rhs.isSubarrayOfArrayAtIndex(cr.rhs, j) then
                newRule = cr.copy().foldAtIndexWithRule(j, r)
                if rules.contains(newRule) then
                    best = nil
                    return
                if newRule.length < best.length then
                    best = newRule
                if newRule.length > 2 then
                    shortenRule(newRule, (byref)best)
                    if best == nil then
                        return
```

Figure 24. The shortening routine

The shortening routine iterates at every position in the rule over a set of rules. The length of the rules depends linearly on the length of the input sentences, i.e. its is $O(n)$, and the number of rules is in the order of $O(s^2n^4)$ which results in a decidedly exponential running time of $O(n^{s^2n^4})$. Nevertheless, I have presented the routine because the general problem of restructuring of rule sets is NP-complete and it seems interesting to investigate whether surrendering to inefficient algorithms leads to drastically improved results. Also, by limiting the folding to singleton rules only the running time is kept in tolerable ranges for many applications.

4.2.3 Experiments

The result of several experiments with the grammars and samples from section 3.4 are shown in figure 25. Obviously, the number of rules is reduced again, albeit not as much as in the last improvement. This is not surprising as only singleton rules could be employed to discover redundancies. However, the main intention of this algorithm was to improve the quality of the rule set by replacing long sequences of words by an appropriate symbol if possible, in order to obtain a more stratified rule set. In this respect, the algorithm is clearly a success as the number of rules which can be shortened is indeed significant. The ambiguity is obviously not affected by the foldings. A complete listing of the rule sets for experiment 2 after various transformations can be found in table 4 in appendix A.

Experiment / Sample	2		3 / #2		3 / #6	
Algorithm	orig.	impr.	orig.	impr.	orig.	impr.
rules	23	22	1 165	968	1 495	1 282
...from contexts	11	10	872	675	1 089	876
...from expressions	12	12	293	293	406	406
...shortened		7		732		978
average ambiguity	∞	∞	20.4	18.2	31.2	28.7

Figure 25. Results of folding singleton rules.

Experiment / Sample	3 / #2				3 / #6			
Improvements	none	U	S	U+S	none	U	S	U+S
rules	1 165	807	968	688	1 495	1 003	1 282	882
...from contexts	872	560	675	441	1 089	677	876	556
...from expressions	293	247	293	247	406	326	406	326
...shortened			732	563			978	727
average ambiguity	20.4	12.9	18.2	11.0	31.2	18.7	28.7	16.8

Figure 26. Comparison and results for the combination of the improvements from section 4.1 (U) and section 4.2 (S).

Both algorithms, i.e. the algorithms from this and the previous section, retain the format of the rule set and it is therefore possible to apply them in sequence. The intuition that the algorithm from section 4.1 should be applied before the algorithm from section 4.2 was verified in several experiments. Figure 26 displays the results for a combined application of both algorithms. The results are encouraging as the number of rules and thus the ambiguity is reduced considerably.

A cursory look at the parse trees for the sentence “John meets the man with the telescope” reveals that by folding the singleton rules, the tendency to produce lists of words has indeed been reduced. The types *CA* and *CN* also show how the heuristic to fold from the back of the sentence produces a linguistically motivated structure. However, as stated before this is a very simple heuristic and nothing much should be expected from it.

```
sntc(john, meets, cn(g(the, man), ca(with, v(the, telescope))))
sntc(john, be(meets, the), man, ca(with, v(the, telescope)))
sntc(john, be(meets, the), ck(man, ca(with, v(the, telescope))))
sntc(john, cr(meets, cc(g(the, man), w(with, the))), telescope)
sntc(john, ba(meets, g(the, man)), ca(with, v(the, telescope)))
sntc(john, bm(meets, p(g(the, man), with)), v(the, telescope))
sntc(cd(john, be(meets, the), man), ca(with, v(the, telescope)))
sntc(bt(john, be(meets, the)), man, ca(with, v(the, telescope)))
sntc(bg(john, meets), g(the, man), ca(with, v(the, telescope)))
```

Figure 27. Parse trees for “John meets the man with the telescope” after removing and shortening rules in the learnt grammar. Compare to figure 19 on page 56.

Generally, one could say that the improvements achieve what was expected from them, but at the same time the result is still anything but satisfactory. If the ambiguity of the parses for this sentence seems tolerable, one should recall that this is the result of a toy problem and not a real world application.

4.3 Type hierarchies

4.3.1 Background

In paragraph 3.2.4 I have described the problem of overlapping sets of expressions and presented a solution for it. This solution was adequate in the sense that it resulted in valid expression types and ultimately in a grammar weakly equivalent to the target grammar. It seems, however, that this solution does not do justice to the linguistic reason for such a situation. Philosophers and linguists, especially those engaged in categorial grammar, have proposed hierarchies for types such as the one sketched in figure 28.

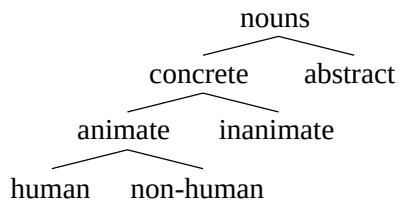


Figure 28. A basic hierarchy for nouns.

Note that the branching is not necessarily binary and hierarchies can be arbitrarily complex as the example from WordNet[2] shows.

```
human, sense 2:
homo, man, human being, human -- (any living or extinct member of the
family Hominidae)
=> entity -- (something having concrete existence; living or
   nonliving)
=> life form, organism, being, living thing -- (any living entity)
=> animal, animate being, beast, brute, creature, fauna -- (a living
   organism characterized by voluntary movement)
=> chordate -- (animal having a notochord)
=> vertebrate, craniate -- (animals having a bony or cartilagenous
   skeleton with a segmented spinal column and a large brain enclosed
   in a skull or cranium)
=> mammal -- (any warm-blooded vertebrate having the skin more or
   less covered with hair; young are born alive except for the small
   subclass of monotremes and nourished with milk)
=> placental mammal, eutherian, eutherian mammal
=> primate
=> hominid
```

Figure 29. A path to the type "human" in the WordNet database.

2 [Miller *et al.*, 93]

Assuming that syntactical structure is not completely arbitrary but to a certain extent influenced by the semantics of the constituents, it is likely that the set inclusions discovered by the Emile algorithm stem from such type hierarchies and it therefore seems logical to extend the algorithm in such a way that it can express type hierarchies.

4.3.2 Description

The problem depicted in figure 9b on page 41 can be generalised to the case where a set of expressions E_A is valid in set of contexts C_A and another set of expressions E_X is valid in a superset of the contexts C_A. See figure 30.

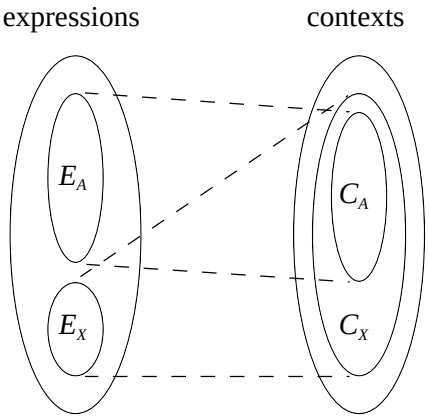


Figure 30. Context set inclusions

The algorithm in its basic form generates from the two expression types *A* and *X* rules stating that each expression in E_A is valid in each context in C_A and rules stating that each expression in E_X is valid in all contexts C_X. The total number of rules is therefore:

$$|E_A \times C_A| + |E_X \times C_X| \tag{4.1}$$

The same situation can be described in a different way, if a new relation between expression types is introduced. In figure 31 the expression type *X* is not defined by the complete set of contexts in which the expressions are valid in but by being a supertype of expressions of type *A* and by the additional contexts.

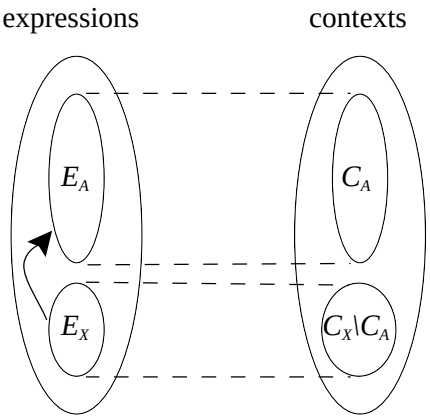


Figure 31. Expression type inclusions

If the expression types A and X are modelled in this way, the algorithm still generates all rules stating that the expressions of type A are valid in contexts C_A. For the expressions in E_X, however, it only creates rules stating that these expressions are valid in the contexts of the set C_X minus C_A. To express that X is a supertype of A it creates one rule, namely $X \rightarrow A$.[3] This results in the following number of rules:

$$|E_A \times C_A| + |E_X \times (C_X \backslash C_A)| + 1 \tag{4.2}$$

Equation (4.1) can be rewritten as follows:

$$|E_A \times C_A| + |E_X \times C_X| \; = \; |E_A \times C_A| + |E_X \times (C_X \backslash C_A)| + |E_X \times C_A| \tag{4.3}$$

Hence, introducing subtypes saves $|E_X \times C_A| - 1$ rules.

If the rule set is modified in the way described above, rules do not only account for syntactic but also for semantic structure. In the parse trees leaves are replaced by paths through the subtype hierarchy as shown in figure 32. This might pose a problem for certain applications but the algorithm can be modified easily to mark rules that are created in this stage, thus allowing a parser to ignore the intermediate nodes.

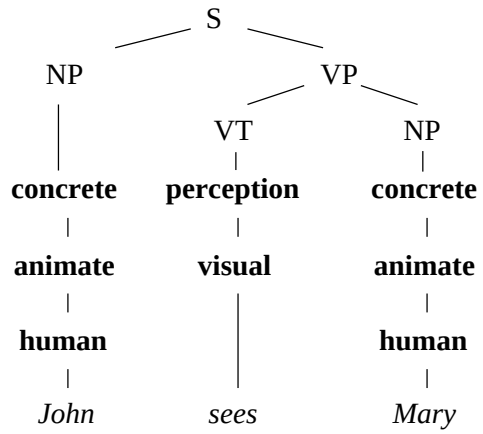


Figure 32. Parse-tree after introducing a type hierarchy.

At this point it is clear how the generalisation stage must be modified if information about type inclusions is present. What is missing is a concept for discovering the hierarchy in the expression types. It might be possible to modify the clustering stage and create hierarchical types right away but for reasons that will become obvious in section 5.1 I choose to implement this improvement in an additional stage after the clustering stage. The procedure works on the types created before and compares for each pair of types the sets of contexts in which expressions of this type are valid in. If one is the superset of the other, the type with the smaller set is added to as subtype to the other type. In the supertype the corresponding contexts are removed. Note that in more complicated situations it must be ensured that only direct subtypes are added. In the

3 With the Lambek calculus in mind the direction of the rule seems to be wrong. However, the rule will be part of a CFG and thus is used in the generating rather than the accepting direction. Its interpretation is that if a symbol X is found, it can be substituted by symbol A and subsequently by all symbols generable from A. And this is exactly what was intended.

case depicted in figure 33 only *B* and *C* are to be added as subtypes to *A* because adding *D* for example would only result in a rule such as $A \rightarrow D$ which is redundant because of $A \rightarrow B \rightarrow D$.

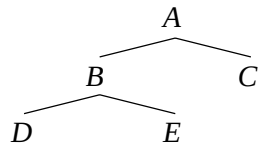


Figure 33. Nested types

As in the other improvements the sequence in which the objects are processed deserves some attention. Consider the case where the algorithm tries to find the subtypes for type *A* first and checks the types in the sequence *D-E-B-C*. After adding *D* and *E* to the list of its subtypes and removing the corresponding contexts, the algorithm discovers that type *B* is also a subtype of *A*. To ensure that *A* is a direct supertype of all of its subtypes it needs to know whether *B* is a supertype of any of *A*'s subtypes. As *B* was not processed yet this information is not available and the algorithm has to postpone processing type *A* and process type *B* first. It eventually discovers that *D* and *E* are in fact subtypes of *B* and therefore has to remove them from the list of type *A* and update the set of contexts accordingly. This might result in a complicated and potentially inefficient algorithm. Fortunately, there is a simple strategy with which the processing can be linearised and updates can be avoided. The algorithm processes the types ordered by the size of their context sets. Multiple sequences are possible because there is no total ordering, but for any sequence it is guaranteed that all subtypes of a potential subtype are known when needed. In the example this means that because the context set of type *B* must be smaller than the one of type *A*, it was checked before *A* and therefore all subtypes of *B* are known when *A* is checked. Moreover, if the algorithm checks for each type the other types in descending order it is ensured that the direct subtypes are found first, i.e. *B* is definitely found before *D* and *E*, which means that updates are avoided. Under these conditions a simple test is sufficient to ensure that only direct subtypes are added: When the algorithm processes a type *X* and encounters a subtype *Y* it traverses the existing subtype hierarchy of type *X* and if *Y* is not found it can conclude that *Y* must be a direct subtype of *X*.

4.3.3 An efficient implementation

First of all the algorithm sorts the types according to the size of their context sets. It then iterates over all types and for each type it iterates over all other types in descending order. It checks whether the context set of the type from the outer loop is a superset of the context set of the other type. If this is the case it checks with a method in the context class whether the other type is already a subtype and if not adds the other type as a subtype. The corresponding method in the context class handles the update of the context set.

To implementation of *hasSubtype* in the context class checks whether the type in question is in its list of contexts. If not it calls each of its subtypes recursively.

Finally, the implementation of the *addSubtype* method in the context class, removes all contexts from its context set and adds the subtype to its list of subtypes. Note the difference between the instance variable *contexts* which holds only the contexts not valid for the subtypes and the method *allContexts* which returns the union of the sets of contexts of all subtypes, i.e. the complete set of contexts in which expressions of the type are valid in.

```
sortedTypes = types.copy().sortBySizeOfContextSet()
foreach ct in { x | x ∈ sortedTypes } do
    foreach t in { x | x ∈ sortedTypes } inReverseOrder do
        if ct.contexts ⊃ t.contexts then
            if not ct.hasSubtype(t) then
                ct.addSubtype(t)
```

Figure 34. Establishing a type hierarchy.

```
boolean Context::hasSubtype(Type t)
    if subtypes.contains(t) then
        return true
    foreach st in { x | x ∈ subtypes } do
        if st.hasSubtype(t) then
            return true
    return false
```

Figure 35. Subtype check in the context class.

```
Context::addSubtype(Type t)
    contexts = contexts \ t.allContexts()
    subtypes.add(t)
```

Figure 36. Adding a subtype to a context.

The number of types is bounded by $O(sn^2)$ and therefore the body of the loops is reached $O(s^2n^4)$ times. The number of contexts is also bounded by $O(sn^2)$ and as it can be determined in $O(m)$ whether a set is proper subset of another set, with m being the size of the smaller set, the subset test takes $O(sn^2)$. For obvious reasons a type can only have $O(sn^2)$ subtypes and thus the recursive test is also bounded by $O(sn^2)$. Finally, removing one context set from another is bounded by the number of contexts. Hence, all operations in the body of the loops are $O(sn^2)$ resulting in a combined running time of $O(s^3n^6)$ for the complete procedure.

4.3.4 Experiments

As in the last sections several of the experiments from section 3.4 were repeated with the improvement presented in this section and figure 37 shows the results. Subtypes are clearly not

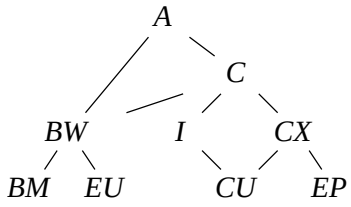
Experiment / Sample	2		3 / #2		3 / #6	
Algorithm	orig.	impr.	orig.	impr.	orig.	impr.
rules	23	22	1 165	797	1 495	1 037
...from contexts	11	9	872	382	1 089	495
...from expressions	12	12	293	293	406	406
...for subtypes		1		122		136
average ambiguity	∞	∞	20.4		31.2	

Figure 37. Results of introducing a type hierarchy.

Experiment / Sample	3 / #2				3 / #6			
Improvements	none	U+S	H	HUS	none	U+S	H	HUS
rules	1 165	688	797	517	1 495	882	1 037	580
...from contexts	872	441	382	182	1 089	556	495	208
...from expressions	293	247	293		406	326	406	
...for subtypes			122				136	
...shortened		563		98		727		95
average ambiguity	20.4	11.0			31.2	16.8		

Figure 38. Comparison and results for the combination of the improvements from section 4.1 (U), section 4.2 (S) and section 4.3 (H).

a reasonable construct for the language from experiment 2. In experiment 3, however, a multitude of nested types is found and the overall number of rules can be reduced. It is interesting to note that the introduction of a type hierarchy results in much fewer foldings if the corresponding algorithm is applied later. See figure 38. This demonstrates that the concept of a type hierarchy is a reasonable one and that folding is a suitable addition for other cases. However, in the present form, the subtype relation is not restricted any further and therefore subtypes are not at all limited to semantic type hierarchies. A part of a type 'lattice' together with the listing of the members of the corresponding types is shown in figure 39.



A = {sees}

BW = {sees the girl with, sees a girl with, sees mary with}

BM = {walks with mary with, walks with john with, finds john with, invites john with, loves mary with, finds a girl with}

EU = {with the flower sees the girl with}

C = {finds, loves}

CI = {sees the girl with the telescope with, sees the telescope with, sees a flower with}

CX = {with a flower finds, with the telescope finds}

CU = {with a flower meets, with the telescope meets, with a telescope walks with, with the flower walks with}

EP = {with a flower finds a girl with, with the flower walks with john with, with the telescope finds john with}

Figure 39. A type 'lattice.'

The combination of all improvements presented so far results in a considerable reduction of the size of the rule set while retaining the language. The latter was ensured for the first two improvements by simply generating the complete corpus and comparing that to the corpus generated by the original grammar. In the case of this improvement the rule set contains several left-recursive rules which makes it impossible to utilise the DCG parser from Prolog. The Rule class was extended such that it could emit a description of itself in a form suitable for a left-corner chart parser and with the help of this parser it was verified that the modified rule set was not

more undergenerating than the original rule set. As chart parsers generally cannot be used as generators and a combination of the Gulp parser with the chart parser failed, no figures regarding the average ambiguity could be obtained.

4.4 Conclusions and future work

4.4.1 Eliminating more structural redundancies

The strategies to remove redundant rules which are described in sections 4.1 and 4.2 do indeed remove a considerable amount of rules but these are chiefly long rules generated by the insertion of an expression type into a context. While this is certainly a desirable result, the main problem, namely the structural redundancy resulting from the structural completeness of the Lambek calculus is not really affected. This is not surprising because a preference was given to shorter rules in the hope to achieve a stratified set of rules rather than a list of possible sentences. This bias, however, hinders the discovery of redundancies in shorter rules as can be illustrated by the following example.

(1) $A \rightarrow$ the girl sees
(2) $A \rightarrow C$ sees
(3) $A \rightarrow$ the D
(4) $C \rightarrow$ the girl
(5) $D \rightarrow$ girl sees

There are obviously three ways in which the expression “the girl sees” can be generated. The algorithm presented in sections 4.1 and 4.2 will remove rule (1) on the basis of rules (2) and (4) or rules (3) and (5). This still leaves two ways in which the expression “the girl walks” can be generated. If the algorithm would prefer longer rules, then rules (2) and (3) could be removed because of rule (1) but as stated before such a bias would result in a tendency to generate lists of expressions. In the light of the current results, however, such an approach seems worth investigating.

Redundancies of the kind described above could also be discovered by unfolding all rules of a given type to a certain depth and comparing the respective extensions. If the extensions resulting from two rules are found to be equal, one of the rules could be removed. At this point, a linguistically motivated choice could be made as to which of the rules should be discarded. I chose to implement the other improvements first because I consider them a necessary prerequisite for the improvement outlined in this paragraph. Note also that unfolding such a large number of rules is an expensive operation and will result in an algorithm with a highly exponential running time. It would nevertheless be interesting to implement this idea because an exponential solution is obviously preferable to no satisfactory solution at all.

4.4.2 Folding with an elaborate bias

Regarding the example from the previous paragraph, the algorithm presented in section 4.2 would also, if rules (2) and (3) were absent, transform rule (1) into one of them. In the actual implementation, preference would be given to rule (3). This is, in this example, obviously a choice against the customary linguistic understanding but as explained in paragraph 4.2.1 on

page 66 the heuristic employed by the algorithm is generally reasonable. One improvement that could be implemented in future versions is to enrich this heuristic with more linguistic knowledge such that sequences of words that are commonly forming phrases are grouped together. In this case preference might be given to a sequence of determiner and noun over noun and verb. This is, however, more complicated than it might seem at first, because in languages with a limited inflectional system, English for example, homonymy is widespread and usually comprises words of fundamentally different classes. If in the example “sees” is replaced by “walks” the algorithm has the choice between a determiner/noun sequence and either a noun/noun or a noun/verb sequence which makes an informed decision almost impossible. Note that the situation cannot be resolved by a part-of-speech tagger[4] because these need a larger context to work reliably.

4.4.3 Learning SCFGs

As it seems difficult or even impossible to eliminate the structural redundancy, one option would be to introduce a measure regarding the ‘usefulness’ of rules. Given such a measure probabilities could be associated with each rule such that rules with a high usefulness would be exponentially or polynomially more probable than less useful rules. If the transformation from the usefulness values to the probabilities is realised in such a way that the sum of the probabilities of all rules of a type is always 1, a SCFG is learned and the algorithms described in section 1.3 could be utilised to establish a ranking of the parses for a sentence and/or expression. This would also leave the option to compare various folding strategies on the basis of the probability that the learned set of rules assigns to a corpus in the target language.

Regarding potential measures for the usefulness of a rule, various options with varying degrees of simplicity and linguistic motivation are available but at the present time further examination is needed to evaluate their usefulness.

4 See paragraph 1.2.4 on page 9.

Chapter

5 Reducing interaction with the oracle

The improvements presented in this chapter aim at reducing the number of questions to the oracle. No attempt has been made so far to change the type of questions, i.e. interaction with the oracle is still limited to retrieving specific positive or negative examples. The improvement described in section 5.1 merges the exploration of the hypotheses space, i.e. stage 2, with the clustering stage. The information obtained is used to prune the hypotheses space while exploring it. In section 5.2 I will describe a more complex data structure to manage the answers from the oracle. It builds on the previous improvement and uses information about types to make correct predictions about answers to questions that have not been asked. In section 5.3 I will investigate how the grammar acquired from a smaller sample can be used in subsequent runs of the algorithm with larger samples in order to reduce the number of questions to the oracle.

5.1 Determining equivalent expressions and contexts

5.1.1 Description

I will describe this and the next improvement with the help of the example from section 3.2. It is therefore necessary to examine how many questions are really asked in the basic version of the algorithm first. If the insertion of an expression into a contexts results in a sentence that is present in the sample no question is necessary. Furthermore, some expression/context pairs result in the same sentence and the answer for the first pair can be used subsequently. Assuming that the substitution matrix is explored top-to-bottom and left-to-right, i.e. by checking one expression in all contexts and then advancing to the next expression, all combinations that are marked with a “Q” in figure 40 do not require a question because they were asked before. All questions to the oracle are marked with a dot. Note that this is a slightly different notation than the one used in chapter 3.

	John	John loves	John loves Mary	loves	loves Mary	Mary	Mary walks	walks
S / loves Mary	S	•	•	•	•	•	•	•
S / Mary	•	S	•	•	•	•	•	•
S	•	•	S	•	Q	•	S	•
John \ *S* / Mary	•	•	•	S	Q	•	•	•
John \ *S*	•	•	Q	Q	S	Q	•	•
John loves \ *S*	•	•	•	•	Q	S	Q	Q
S / walks	•	•	•	•	•	S	•	•
Mary \ *S*	•	•	•	•	•	Q	•	S

Figure 40. The substitution matrix.

The clustering stage of the algorithm finds expression types on the basis of equivalent context sets. If the expressions are checked one by one then the information needed to find most expression types is available before the complete matrix has been explored. In the example the algorithm could determine that “John” and “Mary” belong to the same type, directly after checking “Mary” in column 6. The question is now how the algorithm can make use of this information to reduce the number of questions to the oracle.

If it were known that a bipartite matching existed between the sets of contexts and expressions then upon finding two expressions that are valid in the same contexts all but one of these contexts could be removed from the substitution matrix for the remaining expressions. This would be permissible because the type would partition the set of remaining expressions into two classes: Those that are valid in exactly the same set of contexts and those that are not valid in any of them. To which of these classes an expression belongs can be tested with an arbitrary context of the set. However, it is rarely the case that a bipartite matching exists and even if it does this is not known in advance. Therefore, this idea is not very useful.

By definition the expressions belonging to the same type can be substituted for each other in any context, and not only in the contexts that were explicitly checked. In fact, knowing the latter and asserting the former is one of the inductions employed by the Emile algorithm. Therefore, once several expressions are assigned to the same type, the algorithm can freely exchange them everywhere, even in other expressions and contexts. In the example the algorithm can, after processing the expression “Mary”, replace “Mary” by “John” anywhere. It could replace the expression in column 7 with “John walks” and could also replace the last context with “John \ S.” This last substitution is of particular interest because the resulting context does already exist and therefore one of them can be removed from the matrix for all following expressions. Obviously, the same could happen with expressions resulting in complete columns being discarded.

These observations lead to the improvement proposed in this section. Whenever the algorithm finishes checking one expression it compares the set of contexts this expression is valid in to those of all existing types. If no type with an equal set is found, a new one is created and the expression is made its ‘representative.’ If a type is found then the expression is added to this type and all of its occurrences are replaced by the representative of the type. The latter can result in expressions and contexts to become redundant which means that they can be removed from the matrix. Furthermore, replacing all expression by their representative even if the resulting context or expression does not become redundant is reasonable because using a smaller set of shorter expressions increases the probability that the same sentence is created by several context/expression pairs. The algorithm also employs an optimization that stems from the obvious observation that shorter expressions have a greater chance to occur in contexts or other expressions: Instead of checking the expressions in a random sequence, the algorithm sorts them by their length and starts with the shortest ones.

The result of the improvement is shown in figure 41. For the sake of brevity all words are reduced to their initial letter. After processing column 3 the algorithm replaces every occurrence of “Mary” with “John” as described before. The next type found is {“walks”, “loves John”}, the latter resulting from the original expression “loves Mary.” Now every occurrence of “loves John” is replaced by “walks” which results in another context and the expression in column 8 to become redundant. Note that in this example no further questions could be

	J	L	M		W	J L	L M		MW	JLM
				(M≡J)	W	J L	L J		J W	J L J
								(LJ≡W)	J W	——
S / L M	S	•	•	*S* / L J	•	•	•	*S* / W	•	
S / M	•	•	•	*S* / J	•	S	•	*S* / J	•	
S	•	•	•	S	•	Q	Q	S	S	
J \ *S* / M	•	S	•	J \ *S* / J	•	•	•	J \ *S* / J	•	
J \ *S*	•	•	Q	J \ *S*	Q	•	S	J \ *S*	•	
J L \ *S*	•	•	S	J L \ *S*	•	•	Q	J L \ *S*	•	
S / W	•	•	S	*S* / W	•	Q	•	——		
M \ *S*	•	•	Q	——						

Figure 41. Replacing expressions in the substitution matrix.

avoided inside the matrix, i.e. the replacements did not result in more context/expression pairs forming the same sentence. This effect does become significant in larger matrices, though.

There are two more points regarding this improvement that are noteworthy. I have stated several times that expression types are formed on the basis of equivalent sets of contexts. However, contexts are modified and even removed from the matrix before all expressions are checked which will, given the current state of the improvement, undoubtedly lead to problems. Consider the case illustrated in figure 42 where the sequence of expressions was modified slightly. The expression “walks” is found to be valid in the context set {“John / *S*”, “Mary / *S*”} but when “loves John” is being checked the latter context does not exist any longer, resulting in the context set for this expression to be {“John / *S*”}. The sets are not equal and the expressions cannot be assigned to the same type.

	J	L	W	M		J L	L M		MW	JLM
					(M≡J)	J L	L J		J W	J L J
								(LJ≡W)	J W	——
S / L M	S	•	•	•	*S* / L J	•	•	*S* / W	•	
S / M	•	•	•	•	*S* / J	Q	•	*S* / J	•	
S	•	•	•	•	S	Q	Q	S	Q	
J \ *S* / M	•	S	•	•	J \ *S* / J	•	•	J \ *S* / J	•	
J \ *S*	•	•	•	Q	J \ *S*	•	Q	J \ *S*	•	
J L \ *S*	•	•	•	S	J L \ *S*	•	Q	J L \ *S*	•	
S / W	•	•	•	S	*S* / W	Q	•	——		
M \ *S*	•	•	S	Q	——					

Figure 42. Replacing expressions in the substitution matrix, alternate sequence.

Regarding this example it seems plausible to insert the following operation before checking the equivalence of the set of contexts the current expression is valid in and the set of contexts the expressions of an existing type are valid in. The latter set is intersected with the set of contexts that are present in the substitution matrix, the set of so-called ‘active’ contexts. With this opera-

tion the context “Mary / *S*” would be removed from the context set of “walks” and the sets would be equal. However, contexts are not only removed but also modified, which can also be seen as removing one context and adding another one. After the type {“John”, “Mary”} is found in the example, the context “John \ *S* / Mary” is removed and a completely new context “John \ *S* / John” is added to the matrix. That this can cause problems for the solution described above is obvious. Figure 43 shows part of a substitution matrix for the case that the sample also contains the sentence “John envies Mary.” The expression “loves” and “envies” are of the same type but this would remain undiscovered by the algorithm because of the modification of a relevant context.

	J	L	W	M		E	…
					(M≡J)	E	
S / L M	S	•	•	•	*S* / L J	•	
S / M	•	•	•	•	*S* / J	•	
S	•	•	•	•	S	•	
J \ *S* / M	•	S	•	•	J \ *S* / J	•	
J \ *S*	•	•	•	Q	J \ *S*	•	
…							

Figure 43. Replacing expressions in the substitution matrix, different sample.

To overcome these problems the algorithm maintains a mapping from all contexts that ever existed onto the active contexts and before comparing the context set of an expression to that of a type, both are mapped onto the active contexts. It might seem unnecessary to map the contexts set of the expression because it was only checked in the active contexts. However, the list of valid contexts of an expression also contains those contexts in which the expression occurred in the sentences in the sample and these contexts might have been modified. In the example the data structure for the mappings contains entries such as “John \ *S* / Mary” → “John \ *S* / John” and “Mary \ *S*” → “John \ *S*” which solves both problems described above. This data structure has to be updated in two cases. When a context *B* is found to be equivalent to another context *A* and removed from the substitution matrix because of that, then an arrow $B \rightarrow A$ must be added. As stated before the modification of a context can be understood as removing the original one and inserting a new one. Thus, if the modification of context *B* results in a new context *C* the arrow $B \rightarrow C$ is added.

Finally, it is in most cases better to replace all expressions by the representative of their type but under certain rare circumstances it can happen that an extra question must be asked. In the example the original context “*S* / Mary” is changed into “*S* / John.” Inserting the expression “John loves” into the original context results in a sentence contained in the sample while the sentence resulting from the insertion into the new context is not. In this case no question has to be asked, though, because the sentence “John loves John” was the result of another context/ expression pair. This problem is solved by the improvement in the next section.

5.1.2 An efficient implementation

The algorithm first copies the set of all expressions. Every expression that was checked will be removed from this copy which means that the algorithm has to iterate as long as there are elements left in this set. For each pass through the loop the algorithm first determines the shortest expression. Note that the set of expressions cannot be sorted in advance because the length of the expressions can change when substitutions are made. It can even happen that a shorter expression is checked after a longer one. If, for example, a three-word expression is found to belong to the same type as a one-word expression and the three-word expression occurs twice in a six-word expression the latter collapses into a two-word expression which is checked after the three-word expression. However, this is not problematic as checking the expressions ordered by their length is only an optimization and not a constraint. The selected expression is then removed from the set of expressions that are to be checked.

In the original implementation all contexts were stored in a set. As described above another data structure, i.e. a dictionary providing the mapping from the original contexts onto the active ones, is needed. Instead of maintaining two collections the original set is abolished and for all those contexts that would otherwise not be present in the mapping dictionary because they were

```
expressionsToBeChecked = expressions.copy()
while expressionsToBeChecked.size() > 0 do
    e = expressionsToBeChecked.shortestMember()
    expressionsToBeChecked.remove(e)
    activeContexts = contexts.allValues()
    foreach c in {x | x ∈ activeContexts} do
        s = c.insertExpression(e)
        if answers.containsKey(s) then
            a = answers.valueForKey(s)
        else
            a = oracle.check(s)
            answers.setValueForKey(a, s)
        if answer == 'OK' then
            e.contexts.add(c)
    et = nil
    foreach t in {x | x ∈ types} do
        if t.contexts.map(contexts) == e.contexts.map(contexts)
            et = t
    if(et == nil)
        et = new Type
        et.representative = e
    else
        replaceExpressionWithExpression(e, et.representative)
    et.extension.add(e)
    e.type = et
```

Figure 44. The Emile algorithm, stage 2 and 3 merged

not removed, an entry of the form $C \to C$ is entered. The set of all keys now comprises the set of all contexts. The values of this dictionary, i.e. the right-hand sides of the arrows, are the active contexts and the algorithm iterates over them. The body of this inner loop remains the same as in the original implementation. What follows is the replacement for the clustering stage. The algorithm compares the set of contexts for the current expression with the sets of all existing types, i.e. it compares the appropriately mapped sets with each other. If no suitable type is found, then a new one is created and the expression is made its representative. Otherwise the expression is replaced by the representative of the type in all remaining expressions and active contexts. In any case the expression is added to the extension of the type and the type is noted in the expression.

Regarding the running time not much has changed. The number of expressions and thus the iterations of the outer loop is bounded by $O(sn^2)$. The time needed to find the shortest element is linear in the size of the set, i.e. $O(sn^2)$, which is less than the running time for the inner loop which remains $O(s^2n^4)$. The time needed for mapping the contexts is linearly dependent on the number of contexts which is $O(sn^2)$. However this operation has to be performed for every type and there are $O(sn^2)$ types. This means that this step takes $O(s^2n^4)$ and therefore dominates the body of the outer loop, i.e. unless the routine *replaceExpressionWithExpression* which will be analysed later is more expensive. Assuming this running time for the body of the loop, the running time for this stage is $O(s^4n^8)$, exactly the same as the running time for the original stage 3.

The routine *replaceExpressionWithExpression* takes two arguments, the expression which has to be replaced, named *originalExpression*, and the expression by which it has to be replaced, named *representative*. The first part, shown in figure 45, performs the replacement in all expressions that have not yet been checked. It iterates over these and checks for each whether it contains the original expression. If this is the case, a new expression is created by replacing every occurrence of the words from the original expression with the words from the representative. Note that this operation is ambiguous. If, for example, the original expression is “aba,” the representative is “c” and the expression in which the replacement will be carried out is “ababa” then the result can either be “abc” or “cab.” In the current implementation the choice is left to the corresponding method in the array class. The routine removes the original expression from the set of all expressions and the set of expressions to be checked. If the new expression is not redundant, i.e. not already contained in the set of all expressions, it is added to both sets.

```
foreach e in {x | x ∈ expressionsToBeChecked} do
    if e.words.containsArray(originalExpression.words) then
        newExpression = newExpression
        newExpression.words =
            e.words.replaceOccurancesOfArrayWithArray(
            originalExpression.words, representative.words)
        expressions.remove(originalExpression)
        expressionsToBeChecked.remove(originalExpression)
        if not expressions.containsObject(newExpression) then
            expressions.add(newExpression)
            expressionsToBeChecked.add(newExpression)
```

Figure 45. Replacing an expressions in other expressions

```
newMappings = new Dictionary
foreach c in {x | x ∈ activeContexts} do
    lwords = nil
    if c.leftSide.containsArray(originalExpression.words) then
        lwords = c.leftSide.replaceOccurancesOfArrayWithArray(
            originalExpression.words, representative.words)
    rwords = nil
    if c.rightSide.containsArray(oldExpression.words) then
        rwords = c.rightSide.replaceOccurancesOfArrayWithArray(
            originalExpression.words, representative.words)
    if (lwords != nil) or (rwords != nil) then
        newContext = new Context
        newContext.leftSize = lwords
        newContext.rightSize = rwords
        if not contexts.containsKey(newContext) then
            contexts.setValueForKey(newContext, newContext)
        newMappings.setValueForKey(newContext, c)
foreach c in {x | x ∈ contexts.keys} do
    if newMappings.containsKey(contexts.valueForKey(c)) then
        contexts.setValueForKey(newMappings.valueForKey(
            contexts.valueForKey(c)), c)
```

Figure 46. Replacing an expressions in contexts

The running time for this part is $O(sn^3)$ because there are $O(sn^2)$ expressions and all replacements and checks are bounded by $O(n)$.

The second part of the routine, shown in figure 46, carries out the analogous operation on contexts. This is more complicated because the context mapping must be updated as well. First of all, a dictionary which will hold all mappings resulting from the replacements is created. The routine then iterates over all contexts that are active. For each context it checks whether the original expression is contained in the left and/or right side and in both cases it creates a new side by performing the replacement described above. If a replacement was made a new context is created and added to the global context map if necessary. After processing all active contexts in this manner all new mappings are stored in the dictionary *newMappings* and all new contexts are entered into the global context mapping. However, the global context mapping still has some of the replaced contexts on the right-hand side of its arrows, i.e. it contains pairs of entries of the form $C_a \to C_b$ and $C_b \to C_c$ which would make chained look-ups necessary. To avoid this the former entry will be replaced by $C_a \to C_c$. A more complex case is illustrated in the following example.

contexts before: $\{C_a \to C_x, C_b \to C_b, C_c \to C_x, C_d \to C_y, C_e \to C_z, \ldots C_x \to C_x, \ldots\}$
new mappings: $\{C_x \to C_k, C_y \to C_l\}$
contexts: $\{C_a \to C_k, C_b \to C_b, C_c \to C_k, C_d \to C_l, C_e \to C_z, \ldots C_x \to C_k\}$

Even though the operation seems complicated its implementation is simple. The routine iterates over all keys in the global context mapping. For each key, C_a for example, its checks whether the corresponding value, C_x, is a key in the new mappings. If this is the case, the right-hand side of the new mapping, in this case C_k, is set as the right-hand side in the global mapping.

The number of iterations for both loops is bounded by the number of contexts, i.e. $O(sn^2)$. None of the operations in the bodies of the loops takes longer than $O(n)$ which results in $O(sn^3)$ for this part and also $O(sn^3)$ for the entire routine. This is less than the most expensive operation in the calling loop and therefore the previously stated running time of $O(s^4n^8)$ remains valid.

5.1.3 Experiments

Figure 47 shows the results of the experiments when the contexts and expressions are removed from the substitution matrix as described above. First of all, the desired effect, namely reducing the interaction with the oracle, is achieved and especially for larger substitution matrices the savings are considerable, even though the number of questions remains reasonably large.

Experiment / Sample	2		3 / #2		3 / #6	
Algorithm	orig.	impr.	orig.	impr.	orig.	impr.
questions	99	74	143 979	40 303	318 043	53 044
...relative to matrix	51.6%	38.5%	77.8%	21.9%	79.1%	13.2%
types	7	7	131	129	137	137
rules	23	19	1 165	840	1 495	1 010
undergeneration	0	0	0.06	0	0	0.01
overgeneration	0	0	0	0	0	0
average ambiguity	∞	∞	20.4	12.2	31.2	15.3

Figure 47. Results of removing contexts and expression from the substitution matrix.

It is interesting to consider the effect of the improvement on the learnt grammar. Because a large number of longer expressions is never tested, as these are removed from the matrix, the learnt grammar is more compact. It was foreseeable that this might result in a grammar which is slightly more undergenerating and the run with sample #6 from experiment 3 exhibits this effect. The sentences that cannot be generated are all of the form “John|Mary with the|a telescope meets|invites…” which means that at some point expressions of the form “John with the telescope” were found to be equivalent to expressions such as “John” and the algorithm removed them from the matrix. However, the algorithm represents the subject of a sentence such as “John meets the man…” as a flat list of words. See figure 27 on page 70. This means that “John” is not generated by the symbol which corresponds to the respective expression type and therefore the more complex noun phrase cannot be generated in its place. It is surprising that the opposite effect can occur as well. The grammar learnt by the original algorithm from sample #2 cannot, for example, create any sentence in which a simple-NP plus the adjunct “with the telescope” is the subject of the verbs “finds” and “loves.” The algorithm in this section must have found the corresponding type and must also have created rules in which this type was in the subject position. This can be verified by examining the parse trees for such a

sentence and, indeed, the third tree contains the type *CX* which comprises such NPs in the subject position.

```
sntc(john, ce(with, a, telescope), loves, mary, with, a, flower)
sntc(john, with, a, telescope, cg(loves, mary, with, a, flower))
sntc(cx(john, with, a, telescope), loves, mary, with, a, flower)
```

Figure 48. Parse trees for "John with a telescope loves…"

At the present time it seems to early to determine a general tendency of these effects and therefore one can only conclude that removing expressions from the matrix can result in greater generality but might as well make the discovery of certain constructions impossible.

I will describe the effect of the application of the improvements from the previous chapter as shown in figure 49 together with the results of the next improvement at the end of section 5.2.

Experiment / Sample	3 / #2				3 / #6			
Algor. from section	3.3		5.1		3.3		5.1.	
Restructuring	none	U+S	none	U+S	none	U+S	none	U+S
rules	1 165	688	840	493	1 495	882	1 010	571
...from contexts	872	441	682	336	1089	556	842	403
...from expressions	293	247	158	157	406	326	168	168
...shortened		563		337		727		430
average ambiguity	20.4	11.0	12.3	7.2	31.2	16.8	15.3	8.8

Figure 49. Results of the application of the improvements from sections 4.1 and 4.2 to the rule set learnt by the original algorithm and the one presented in section 5.1.

5.2 Inducing equivalence classes of questions

5.2.1 Description

The improvement in the last section uses the information about type membership to prune the substitution matrix. In this section the data structure in which the answers from the oracle are stored is modified such that type information can be used to make correct predictions for answers to questions that have not been asked. The idea for the improvement can be outlined easily. Suppose that "John" and "Mary" are known to be of the same type. If the algorithm needs to know whether the sentence "Mary loves John" is correct and the sentence "John loves Mary" has been accepted by the oracle then no question has to be asked. Moreover, the assertion regarding expression types has so far been formulated in an overly strict way. It was said that expressions of the same type can be exchanged without destroying grammaticality. The assertion is, in fact, more general and could be modified such that grammaticality is not affected by exchanging expressions of the same type, i.e. ungrammaticality is preserved as well as grammaticality. Given this clarification it becomes obvious that not only the few questions which will be answered positively but also the large amount of questions with negative answers benefit from this improvement. If, for example, the algorithm constructed the sentence "Mary loves loves" but the oracle has rejected "John loves loves" before, the answer is predictable. In

figure 50 additional questions that can be avoided are marked with an “E.” For example, number 1 results in “John walks John” which is equivalent to “John loves John John.”

	J	L	M		W	J L	L M		MW	JLM
				(M≡J)	W	J L	L J		J W	J L J
								(LJ≡W)	J W	——
S / L M	S	•	•	*S* / L J	•	•	•	*S* / W	•	
S / M	•	•	•	*S* / J	•	Q	•	*S* / J	E[1]	
S	•	•	•	S	•	Q	Q	S	Q	
J \ *S* / M	•	S	•	J \ *S* / J	•	•[2]	•[1]	J \ *S* / J	•	
J \ *S*	•	•	Q	J \ *S*	Q	•	Q	J \ *S*	E[2]	
J L \ *S*	•	•	S	J L \ *S*	•	•	Q	J L \ *S*	•	
S / W	•	•	S	*S* / W	•	Q	•	——		
M \ *S*	•	•	Q	——						

Figure 50. Predicting answers in the substitution matrix.

Regarding the algorithmic realisation of this idea it might seem sufficient to replace every expression by the representative of its type in all questions asked so far. If “Mary” is assigned to the same type as “John” then the question “John loves Mary” would be changed into “John loves John.” This seems sufficient because “Mary” is also replaced by “John” in all contexts and expressions which remain to be checked and therefore all future questions will also contain “John” instead of “Mary.” However, this procedure does not work for multi-word expressions which is illustrated in the following example.

expressions = {loves, John, walks, …loves John, John loves, the girl, girl loves, …}

contexts = {John \ *S* / John, The \ *S* / John, *S* / John, …}

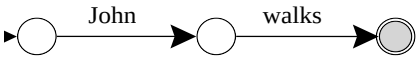
Inserting “loves” into the context “John \ *S* / John” results in the sentence “John loves John” which is accepted by the oracle and stored in the list of known answers. When the algorithm discovers that “loves John” is of the same type as “walks” it would also replace the sentence “John loves John” by “John walks” in its answer dictionary. When it then checks whether the expression “John loves” and “*S* / John” form a correct sentence, the previous answer is not available anymore and the question has to be asked again. An obvious countermeasure would be to keep both sentences in the answer dictionary. There is, however, another similar shortcoming in this approach. Continuing in the example the algorithm finds out that “John” and “the girl” belong to the same type and updates its data structures accordingly. Later, it inserts the expression “girl loves” into the context “The \ *S* / John” which results in the sentence “The girl loves John.” Assuming that this sentence has not been the result of another pair and given the current types, the answer could be inferred from the question “John loves John.” So far, multi-word expressions are kept in the answer dictionary and they are also replaced by their representative. Now it would be desirable to replace the representative by the multi-word expression. Furthermore, it is also possible to construct examples in which one multi-word expression of a type should be replaced by another multi-word expression of the same type, or in which the result of a replacement is the subject of another replacement.

It is not possible to use the information about types to create all possible sentences in advance because their number grows exponentially in the number of expressions. Another approach

could be to consider the equivalency of expressions of the same type as rules of a grammar and then attempt to derive any of the known questions from the candidate question. A brief look at some of the rules that would be generated in the example reveals a problem with this approach.

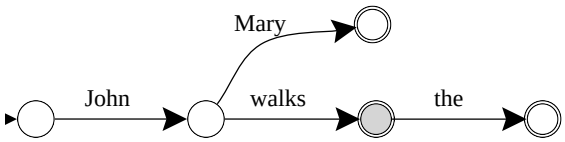
walks → John loves	John loves → walks
the girl → John	John → the girl

These rules are not context-free but context-sensitive and therefore a parser would require exponential time. However, understanding the problem as the word problem for some language is the key to a solution. The larger part of the language of questions with predictable answers can be generated by a DFA and can be represented as a directed graph.



Words are located on the edges. There is one designated start state and any number of accepting states, marked by a double circle. Accepting states carry information whether the corresponding sentence is correct, shown by a shaded circle in the diagrams, or not acceptable, shown by a plain circle. To determine whether a sentence is correct it is sufficient to begin at the start state and follow the edges. If an accepting state is reached the answer can be read from it. Otherwise, the routine returns that no answer is known which will lead to an oracle query.

When a new answer is to be inserted the routine follows the edges as long as possible and then adds new vertices and edges as necessary. The following figure shows the graph after entering the sentences “John Mary” and “John walks the” with negative answers.



Both operations are simple and linear in the length of the expression. Noting the equivalence of two expressions is more complicated and results in a variety of cases, some of which are depicted in figure 51. At first, the equality of “John” and “Mary” is noted. Since both are one-word expressions the new member of the type, namely “Mary,” is simply replaced by the representative of the type, “John.” That this is a reasonable and legal proceeding was explained before. The second diagram shows the graph after entering the equality of “loves John” and “walks.”

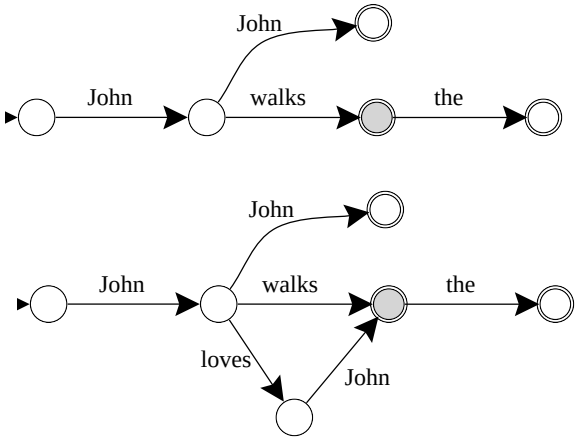


Figure 51. Establishing equalities in the question graph.

The examples in figure 51 represent simplified cases because the expressions found to belong to an existing type were not present in the graph before which is impossible in real applications. For example, to find out that “loves John” is of the same type as “walks” a question including “loves John” must have been asked before and therefore the expression must be present in the graph when the equality is noted. This means that instead of adding new edges, existing states or even subgraphs must be merged. A simple case is depicted in figure 52. Noting that the expressions “loves John” and “walks” are of the same type means that states 3 and 5 are equivalent.

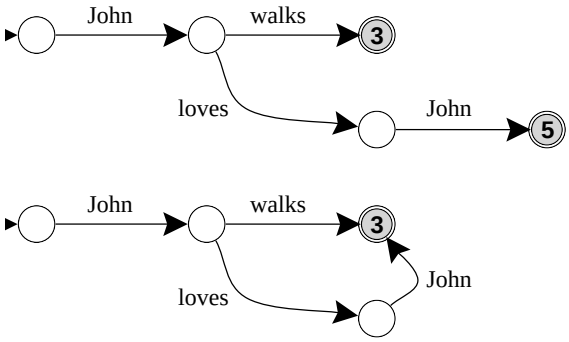


Figure 52. Merging states in the question graph.

In the last example both nodes were terminal nodes (and accepting states but that is not relevant.) If both of the nodes have successors then the corresponding subgraphs have to merged, i.e. the successors of the merged node are the ‘union’ of the successors of the individual nodes. Figure 53 illustrates this. Note in particular that not only the edges but also the acceptance status of the vertices is transferred. Before the merge operation it was not known whether the sentence “John walks the” is correct, but since “John loves John the” was known to be incorrect this information is available after the merge.

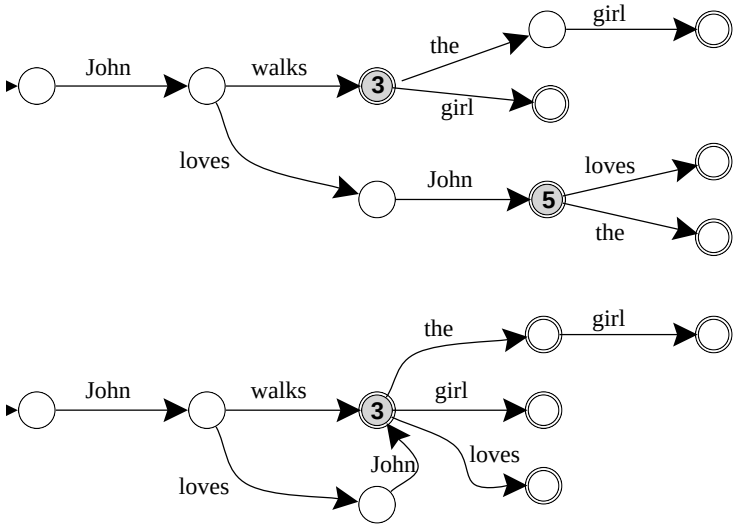


Figure 53. Merging subgraphs in the question graph.

The final case is that of multi-word expressions which contain the representative of their type. If the representative is a prefix of the expression a cycle will be added to the graph as shown in

figure 54 where the equivalence of “John” and “John with the beard” is processed. The same holds true for postfixes. If, however, the representative is located in the middle of the expression as in “abc” and “b” then the information contained in this equivalency cannot be represented properly. This is not surprising because the extension of this type is $\{a^n bc^n \mid n \geq 1\}$ which is not regular and thus not generable by a DFA. As stated before the problem is located in the class of context-sensitive languages and using a much weaker formalism must inevitably result in cases that cannot be handled. In most practical applications these cases are rare and the gain in efficiency for the remaining cases clearly compensates for this disadvantage.

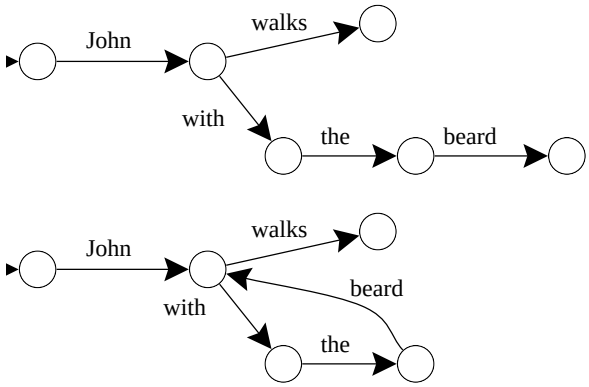


Figure 54. Cycles in the question graph.

5.2.2 Merging subgraphs step-by-step

In this paragraph I will present the algorithm to merge subgraphs by means of an extended example that illustrates most cases. Figure 55 shows an example graph and the result after noting the equivalence of “loves Mary” and “walks.”

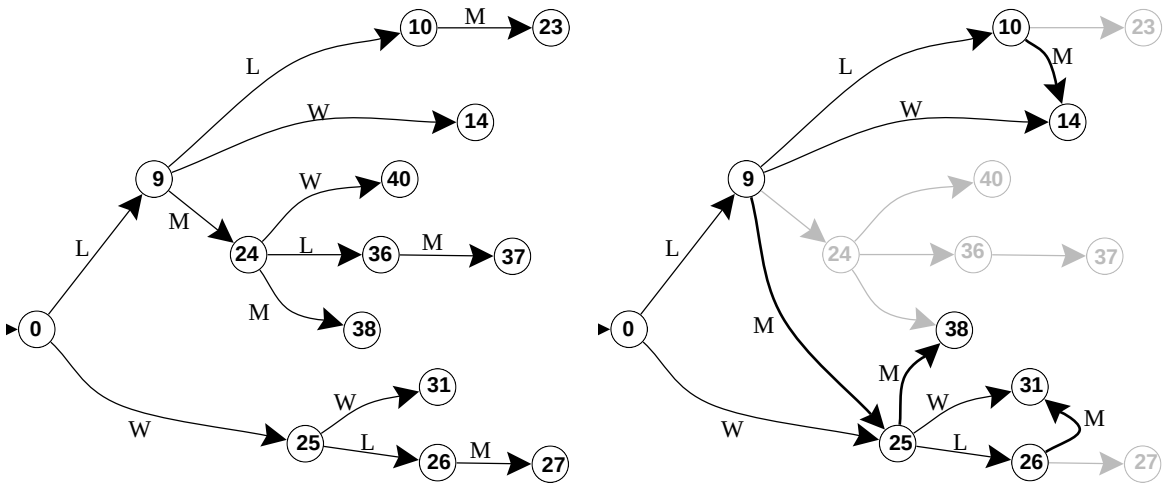


Figure 55a and b. Merging large subgraphs in the question graph.

The algorithm traverses through the graph with a depth-first strategy. For each vertex that is reached I will state the number of the vertex, the path that was taken to reach the vertex, a canonical path and the partial sentence that was formed so far. The canonical path is the path

starting at the root following the representatives of the types of the expressions on the path to the current vertex. The canonical path is empty when it is equivalent to the normal path.

no.	path	canonical path	partial sentence
0	0	–	ε
9	0, 9	–	L
10	0, 9, 10	–	L L
23	0, 9, 10, 23	0, 9, 14	L L M

The expression “loves Mary” is found at the end of the partial sentence and the canonical path has to be calculated. The algorithm copies the normal path and removes as many elements from its end as the expression which is to be replaced has words, two in this case. From the last vertex on that path it follows the edge for the representative and reaches vertex 14. In this case the complete canonical path exists already and therefore all references to the current vertex can be redirected to the end of the canonical path. The edge between vertices 10 and 23 is removed and an edge from 10 to 14 is added. Vertex 23 is deleted subsequently because it is not referenced any longer.

0	0, 9, 14	–	L W
9	0, 9, 24	0, 25	L M

The expression is found again and the canonical path exists. Edge $\langle 9, 24\rangle$ is replaced by edge $\langle 9, 25\rangle$ and vertex 24 is deleted because it is unreferenced now. Its edges are still being followed, though.

40	0, 9, 24, 40	0, 25, 31	L M W

The expression is not found but the canonical path is not empty. Therefore, the algorithm tries to follow from the last vertex on the canonical path the edge for the last symbol on the normal path. This is possible in this case and vertex 31 is reached. Vertex 40 is deleted because it is not referenced by any vertex.

36	0, 9, 24, 36	0, 25, 26	L M L
37	0, 9, 24, 36, 37	0, 25, 31	L M L M

The expression is found and the canonical path is not empty. The algorithm removes n elements from the canonical path, where n is the number of words in the expression minus 1, and from there tries to follows the edges for the representative. This is possible and leads to vertex 31. Again, no references are left to vertex 37 and it is deleted.

38	0, 9, 24, 38	0, 25, 38	L M M

The expression is not found but the canonical path is not empty. The algorithm tries to follow from the last vertex on the canonical path the edge for the last symbol on the normal path which is not possible in this case. The furthest the algorithm can get is to vertex 25. It would now start adding new edges and vertices until it reaches the second last symbol. In the example it was able to reach the second last symbol so this step is not necessary. It then adds an edge from the

last vertex on the canonical path to the last vertex on the normal path, ⟨25, 38⟩. Therefore, vertex 38 still has a reference and is not deleted like the other successors of vertex 24.

25	0, 25	–	W
31	0, 25, 31	–	W W
26	0, 25, 26	–	W L
27	0, 25, 26, 27	0, 25, 31	W L M

The expression is found once more and, again, the canonical path already exists in the graph. Edge ⟨26, 37⟩ is replaced by ⟨26, 31⟩ and vertex 27 is deleted.

The graph in the example consisted of tree edges only. Generally, it cannot have any cross edges because the complete graph is always rooted in one designated start state but it can have forward and back edges. Normally, a depth-first search procedure stops when it reaches a vertex that has been visited before, i.e. after following a forward or back edge. The merging algorithm has to continue in this case because a multi-word expression can span a forward edge or it might be the case that the expression can only be formed by following a cycle a number of times. However, it can obviously not follow cycles indefinitely. The algorithm always maintains a count of how many words of the expression have been matched so far. When it follows a back edge and no word has been matched it can stop because any match that could occur from then on could have occurred without following the cycle. If at least one word has been matched the algorithm sets a counter to the number of words remaining in the expression. At every new vertex this counter is decreased by one and if it reaches zero traversal is stopped. If a mismatch is detected earlier traversal is obviously stopped immediately.

Note that in the example no attention was paid to the acceptance status of the vertices. Whenever the edges from a vertex are redirected to the corresponding canonical vertex and the vertex represents an accepting state then the answer is copied into the canonical vertex. If the oracle is consistent then the canonical vertex cannot already contain a contradictory answer. In the implementation a contradiction is resolved by discarding both answers for the current vertex, thus forcing another question for this sentence.

5.2.3 An efficient implementation

The graph is represented using an adjacency list data structure. Each vertex is represented by a vertex object which contains a dictionary with the words of the outgoing edges as keys and the target nodes as values. Vertex objects also maintain a list containing all references to them, including the referencing vertex and the corresponding word.

It would not only be inefficient to compare for each node visited whether the last symbols of the partial sentence are equal to the expression which is to be replaced, but it is also insufficient for the algorithm because in order to handle back edges properly it must be known at any time how much of the current expression has been matched already. Furthermore, an expression can contain prefixes of itself which means that when after matching a part of the expression a mismatch is encountered it is not possible to simply restart the match at the beginning of the expression. Given that n words have been matched, a naive solution would be to go back n–1 words and start matching at the beginning of the expression. This, however, would severely complicate the traversal of the graph. On the other hand, matching the expression with the partial sentence can

be seen as a string matching problem where each word corresponds to a symbol and the alphabet is the union of all words in the sample. It is therefore possible to use the Knuth-Morris-Pratt string matching algorithm[1] to find the expression. This algorithm computes a so-called prefix function in advance that makes it possible to find all occurrences of a string *p* in another string *t* by passing through the string *t* once without backtracking. When this algorithm is embedded in the depth-first traversal, it can be decided at each vertex in $O(1)$ whether the expression was completely matched and how many symbols are still matched when a mismatch is encountered.

The routine called from the main algorithm takes as parameters an expression *ce* which has been assigned to a type *t* and the representative *r* of that type. It computes the prefix function for *ce* and then calls a recursive method with certain initial values to traverse the graph.

```
prefixFunction = calculatePrefixFunction(ce)
path = new Array
path.add(root)
noteEquivalence(ce, prefixFunction, r, new Set,
                                        path, nil, new Array, 0, -1)
```

Figure 56. noteEquivalence initialiser routine

The recursive method takes as parameters:

- *ce*, the expression which has been assigned to a type.
- *prefixFunction*, the prefix function for *ce* according to the KMP algorithm.
- *r*, the representative of the type.
- *visited*, a set containing all vertices that have been visited. A vertex is added to this set when it is reached first. This set is used to detect forward and back edges.
- *path*, an array containing the objects on the path in the order they were reached.
- *canonicalPath*, an array containing the canonical path as defined above.
- *sentence*, an array containing the sequence of words formed by the edges on the path.
- *matchlen*, the length of the longest prefix of *ce* which is equal to a postfix of *sentence.*
- *ttl*, the maximum depth including the current vertex or –1 if no depth limit is in effect. This is used to limit the depth after a back or forward edge has been crossed.

The sets, paths and sentence variables are not copied when the routine calls itself recursively. This means that the callee shares the collection object with the caller and has to undo all modifications that are carried out during its invocation before it returns. This might seem awkward but it would be too expensive to copy the potentially large collections on every call of the routine.

1 [Kntuh *et al.*, 77]

First of all, the routine determines from the paths the current vertex *vertex* and the canonical vertex *canonVertex*. It also copies all references from the current vertex into another dictionary.

```
vertex = path[path.length -1]
canonVertex = canonPath[canonPath - 1]
originalReferences = vertex.references.copy()
```

Figure 57. noteEquivalence routine, initialisation

The first operation is adjusting the match length. The routine gets the last word of the sentence formed by the expressions on the current path and compares it to the next word in the expression. If they are equal then the match length is incremented. Otherwise the routine determines with the help of the prefix function the number of words of the expression that can still be matched because the expression contains a prefix of itself. Say, the expression is "ababc" and the sentence is "…ababa". In this case *matchlen* would be 4 because the first four words of the expression matched the end of the sentence up to and including the second last word. The routine would now check whether the next word of the expression, i.e. "c" is equal to the last word of the sentence, "a" in this case. Since they are not equal the match in progress is invalid but another, shorter match can also be considered in progress, namely matching the first two letters of the expression to the end of the sentence. According to that match the next word to be matched in the expression is "a" which is equal to the last word of the sentence and therefore the match would continue with *matchlen* being 3. If no such restarting point is found in the expression *matchlen* is set to zero with the obvious meaning that nothing was matched so far.

```
if sentence.length > 0 then
    lastWord = sentence.words[sentence.words.count() - 1]
    if ce.words[matchlen] == lastWord then
        matchlen = matchlen + 1
    else
        while (matchlen > 0) and (ce.words[matchlen] != lastWord) do
            matchlen = prefixFunction[matchlen - 1]
```

Figure 58. noteEquivalence routine, adjusting the match length.

When *matchlen* is equal to the length of the expression then a complete match has been found. At first it is checked whether the equivalence is between to single-word expressions and in this case the edge connecting the second last vertex on the path and the current vertex with the expression is removed. This is important because later in the procedure all references to the current vertex will be redirected to the canonical vertex but all occurrences of expressions of this kind should be removed from the graph. By handling this case at this place later operations are simplified. If the variable for the canonical path is nil this means that the canonical path is identical to the current path. As this is going to change the variable is set to the current path. Note, that all modifications have to be undone before the method returns. In this case, however, it would be complicated to track all modifications of the canonical path and therefore the array representing the canonical path is copied so that modifications do not affect the array that the caller uses.

Now, the canonical vertex corresponding to the current vertex must be found. This is done by removing the expression from the canonical path and adding the vertices which are found by

following the words of the representative from this point. If a vertex on the canonical path does not exist it is created. If at the second last vertex of the canonical path no edge corresponding to the last word of the representative is found the current vertex is re-used as the canonical vertex. This is legal because at this point the creation of a new vertex would result in redirecting all references to the original vertex to the new vertex and all vertices beyond the current one would be added beyond the new vertex. The same is achieved by using the current vertex as canonical vertex. In this case, and any other case which results in the current vertex being the canonical vertex the variable for the canonical vertex is set to nil. Finally, the current match length is updated.

```
if matchlen == ce.words.count then
    if expr.words.count() == 1 then
        path[path.length - 2].removeEdgeForWord(ce.words[0])
    if canonPath == nil then
        canonPath = path
    canonPath = canonPath.copy()
    for i = 0 to ce.words.count() do
        canonPath.removeLastObject()
    canonVertex = canonPath[canonPath.length - 1]
    for i = 0 to ce.words.count() - 1 do
        w = ce.words[i]
        nextCanonVertex = [canonVertex vertexForWord:w]
        if nextCanonVertex == nil then
            if i == ce.words.count() - 1 then
                nextCanonVertex = vertex
            else
                nextCanonVertex = new Vertex
            vertex.addEdge(nextCanonVertex, w)
        canonVertex = nextCanonVertext
        canonPath.add(canonVertex)
    if canonVertex == vertex then
        canonVertex = nil
    matchlen = prefixFunction[matchlen - 1]
```

Figure 59. noteEquivalence routine, determining the canonical path after a match.

The next step in the routine is to redirect all references from the current vertex to the canonical vertex. Note that a canonical path can be passed in from the caller and therefore this operation must always be performed and not only if a match was found. All referencing vertices together with the edge that connects them to the current vertex are determined. The edges are removed and new edges are created between each of the referencing vertices and the canonical vertex. Afterwards all edges from the current vertex to other vertices are deleted. Note that a copy was made before so that they can still be followed in the next step. The final part of this operation is to update the answer of the canonical vertex. If the current vertex contains an answer but the canonical vertex does not, the answer is copied. If both vertices contain an answer these should in theory always be identical. However, in practice inconsistent answers of the oracle, which

```
if canonVertex != nil then
    foreach v in {x | x ∈ vertex.referencingVertices()} do
        word = v.wordOnEdgeToVertex(vertex)
        v.removeEdgeToVertex(vertex)
        v.addEdge(word, canonVertex)
    foreach v in {x | x ∈ vertex.referencedVertices()} do
        vertex.removeEdgeToVertex(v)
    if vertex.answer != nil then
        if canonVertex.answer != nil then
            if vertex.answer != canonVertex.answer then
                vertex.answer = nil
        else
            canonVertex.answer = vertex.answer
```

Figure 60. noteEquivalence routine, redirecting references to the canonical vertex.

might also result from too small samples in cases where a parser is used, do sometimes occur. In this case the answer is deleted and thus it is left to the oracle to decide again.

At this stage a match of the expression has been processed, the canonical path has been adjusted and, if necessary, the current vertex has been replaced by its canonical counterpart. Now, all references from the current vertex have to be traversed recursively. First, the current vertex is added to the set of visited vertices so that other edges pointing to it can be identified as forward or back edges. Then it is checked whether a time to live is set because a forward or back edge has been crossed before. If this is the case and the length of words to be matched is larger than the time to live, probably because a mismatch occurred, traversal is stopped immediately by returning from the method. Otherwise the algorithm collects all words on the edges leaving the current vertex in a set named *allWords*, iterates over these and removes every word, i.e. edge that has been processed. Consequently, the loop is terminated if this set is empty.

The algorithm tries to follow the edge matching the next word in the expression. If this does not exist or has been processed an arbitrary edge is chosen. The motivation for this behaviour is that this will with a greater probability result in the canonical path to be ‘below’ the current path and therefore new combinations that arise because of merging vertices with the canonical vertices can be exploited. After an edge is chosen the word is removed from the set and the next vertex is determined. If the algorithm is about to follow a back or forward edge, i.e. the next vertex is in the set of visited vertices, or the algorithm has done so, i.e. time to live is not –1, it checks whether the word on the current edge is equal to the next word which has to be matched. Failing this test the algorithm can continue the loop with the next word because following this edge to a vertex that has been visited before without a match in progress would result in the same situation as before. The next step is to determine the next canonical vertex if one is set. If an edge with the next word exists at the current canonical vertex then its destination is the new canonical vertex. Note that even if the word corresponds to a one-word expression that is to be replaced and must not occur in the graph after traversal is complete, the edge can be used, because the canonical vertex which it is added to is guaranteed to be visited after the current vertex due to the selection of the edges above. Therefore, the edge will be removed later. If the canonical vertex contains no suitable edge then an edge can be added from the last canonical

```
visited.add(vertex)
if (ttl != -1)or (ttl < expr.words.count() - matchlen) then
    return
allWords = originalReferences.keys
while allWords.count() > 0 do
    if allWords.contains(ce.words[matchlen]) then
        word = ce.word[matchlen]
    else
        word = allWords.anyElement()
    allWords.remove(word)
    nextVertex = originalReferences.valueForKey(word)
    if (visited.contains(nextVertex)) or (ttl == -1) then
        if word != ce.word[matchlen] then
            continue with next word
    if ttl != -1 then
        nextTTL = ttl - 1
    else if visited.contains(nextVertex) then
        nextTTL = ce.words.count() - matchlen
    else
        nextTTL = -1
    nextCanonVertex = nil
    if canonVertex != nil then
        nextCanonVertex = canonVertex.vertexForWord(word)
        if nextCanonVertex == nil then
            canonVertex.addEdge(word, nextVertex)
    sentence.add(word)
    path.add(nextVertex)
    if(nextCanonVertex != nil) and (nextCanonVertex != nextVertex)
        canonPath.add(nextCanonVertex)
        noteEquivalence(ce, prefixFunction, r, visisted, path,
            canonPath, sentence, matchlen, nextTTL)
        canonPath.remove(nextCanonVertex)
    else
        noteEquivalence(ce, prefixFunction, r, visisted, path,
            nil, sentence, matchlen, nextTTL)
    path.removeLast()
    sentence.removeLast()
```

Figure 61. noteEquivalence routine, traversing all referenced vertices

vertex to the next vertex, thus rejoining the paths. Obviously, the next canonical vertex is nil in this case. After these preparations it is checked whether a next canonical vertex is set, i.e. the current path and the canonical path could not be joined. In any case the current word is appended to the sentence and the current vertex is added to the path. The same is done for the canonical path if necessary. Then, the method calls itself recursively to process one of the verti-

ces following the current one. When this call returns the word and vertices are removed from the sentence and paths.

As this method is called from stage 3 of the Emile algorithm whenever an expression is found to belong to a type, an exact worst-case analysis of the algorithm alone is not very meaningful. It would be interesting to know what the worst-case running time of stage 3 is when this data structure is used, but the interaction between the main loop of stage 3, i.e. the number of types found from the expressions and contexts and the structure of the graph is so intricate that is seems to be beyond the scope of this thesis to make a reasonable estimate for the overall running time of a modified stage 3. Furthermore, due to the mode of operation it can be expected that the worst-case running time of this algorithm differs greatly from the average case which is the case that is important for a practical application. However, I will outline why the running time of stage 3 using this data structure to manage the answers is polynomially bounded, thus showing that it can be considered efficient in the strict sense.

The size of the substitution matrix is $O(s^2n^4)$ and for each entry a sentence with n words is stored in the graph. The costs of the insertion and retrieval operations are linear in the length of the sentence. This means that under no circumstances the graph can contain more than $O(s^2n^5)$ vertices. The number of calls of the *noteEquivalence* method is obviously bounded by the number of expressions, i.e. $O(sn^2)$. It is therefore sufficient to show that the number of operations of the algorithm to merge the subgraphs is polynomially bounded in the number of vertices v in order to show that the overall running-time of stage 3 is polynomially bounded.

The number of operations of a normal depth-first search algorithm is bounded by $O(v+e)$. Also, the number of edges e is $O(v^2)$ as each vertex can be connected to each other vertex. In the case of the answer graph the number of edges is $O(vsn)$ because each vertex can only have as many outgoing edges as there are different words in the sample. The algorithm visits each vertex once by following the tree edges. Because the graph is loosely connected, i.e. for each pair of vertices $\langle x, y\rangle$ either x is reachable from y or vice versa, there are exactly $v - 1$ tree edges. If the algorithm encounters a forward or back edge it visits from there on at most l vertices and edges, where l is the length of the expression. As stated before the length of an expression is $O(2n)$. Given, that all operations in one call of the function are polynomially bounded, a fact that can be verified easily, and that there are f forward edges the running time of this algorithm is $O(v+nf)$. The number of forward edges is obviously bounded by the number of edges that are possible in a directed graph, i.e. $O(v^2)$, but in practical applications the number of forward edges is usually much smaller than the number of tree edges.

5.2.4 Experiments

As the algorithm presented in this section is a direct extension of the one from the previous section, the results are not shown in relation to the original algorithm but in relation its antecedent. See figure 62. As expected the number of questions is lower again. Furthermore, the tendency of the last algorithm to generalise is further increased. This is caused by cycles in the answer graph, because adding a back edge in order to create a certain path in the graph can also result in the addition of further paths which were not intended. In both samples of experiment 3, the algorithm inadvertently creates paths in the answer graph such that INSTR adjuncts are not only allowed for the verb “sees” but for all other verbs as well. This also explains why the overgeneration is comparatively high. As explained before[2] overgeneration is not harmful for certain applications and in such cases the improvements offered by this algorithm are clearly

desirable. If overgeneration cannot be allowed then this algorithm cannot be used because the cause for this problem is directly linked to the structure of the algorithm. It ultimately results from the fact that a formalism corresponding to regular languages is used for a problem that can only be expressed with context-sensitive languages.

Experiment / Sample	2		3 / #2		3 / #6	
Algor. from section	5.1	5.2	5.1	5.2	5.1	5.2
questions	74	73	40 303	32 875	53 044	44 208
...relative to matrix	38.5%	38.0%	21.9%	17.9%	13.2%	11.0%
types	7	7	129	126	137	131
rules	19	19	840	855	1 010	1 000
undergeneration	0	0	0	0	0.01	0.01
overgeneration	0	0	0	0.62	0	0.60
average ambiguity	∞	∞	12.2	8.7	15.3	9.97

Figure 62. Results of removing contexts and expression from the substitution matrix and inducing equivalence classes of types.

When the restructuring algorithms are applied to the grammar learnt by the algorithm from this section, comparatively reasonable grammars are the result. As all grammars for experiment 3 are prohibitively large, I have included a listing of the original grammar for experiment 2 and the grammar resulting from the combination of the improvements in sections 4.1, 4.2, 5.1 and 5.3 in table 4 in appendix A. Figure 63 gives an overview of the results for experiment 3, a full listing is shown in table 2 in appendix A.

Experiment / Sample	2		3 / #2		3 / #6	
Restructuring	U+S	H+U+S	U+S	H+U+S	U+S	H+U+S
rules	14	16	512	465	575	501
...from contexts	5	6	156	210	408	231
...shortened	6	6	399	104	433	96
average ambiguity	∞	∞	5.3		5.9	

Figure 63. Application of the improvements from sections 4.1 and 4.2 to the grammar learnt by the algorithm in section 5.2

Recalling that the number of rules of the grammar for sample #6 as learnt by the original algorithm is 1495 and the average ambiguity is well over 30, then the results achieved by the various improvements seem considerable. But also recalling the fact that all the examples are from a toy problem it is in my opinion too early to consider any of the improvements a breakthrough.

2 See the discussion on page 9.

5.3 Other options

5.3.1 Incremental versions

The algorithms presented in the previous two sections reduced the number of oracle queries for a given substitution matrix considerably. However, the reductions are not in the order of magnitudes and therefore the size of the sample can still not be increased as much as it would be necessary for interesting target languages, i.e. languages as they occur in real applications. It is therefore necessary to investigate completely different strategies. So far, all improvements aimed at avoiding questions during the exploration of the substitution matrix. No attempt was made to reduce the size of the matrix before exploration starts. At the end of paragraph 3.5.2 I indicated that such a solution seems possible with an incremental version of the algorithm that splits the sample into several batches and processes each of them individually. The first batch would contain the least complex sentences according to some constructive criterion and subsequent batches would comprise sentences that exemplify constructions of increasing complexity. In this case the sum of the sizes of the smaller substitution matrices would indeed be polynomially smaller than the size of the complete substitution matrix. Furthermore, such an algorithm would always leave the option to start with a small hand-written core grammar. Figure 64 summarizes these ideas. On the left, questions that can be avoided due to the improvements presented in the previous section are marked in gray. On the right, the substitution matrices resulting from breaking up the sample are shown in the original substitution matrix. Utilising the results from the previous runs, the areas delimited by the dashed lines can be considered explored.

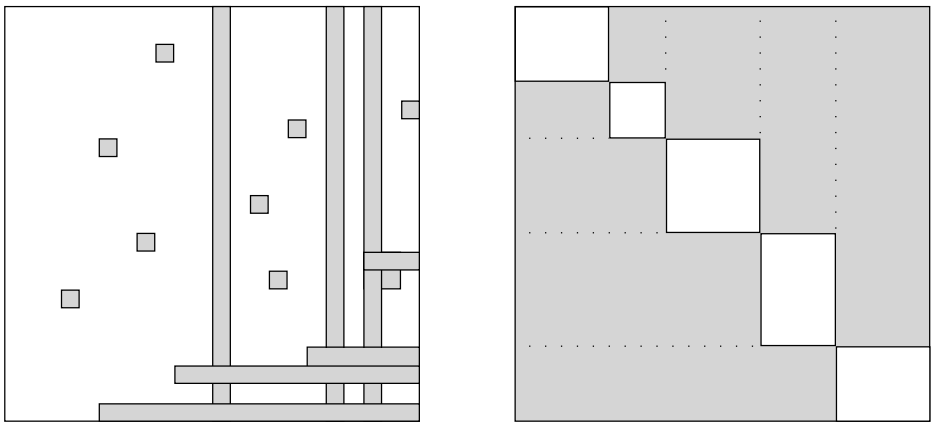


Figure 64. Strategies to reduce the number of questions.

While there is no doubt that an incremental version would be superior to the improvements presented so far, it is also clear that there are at least two problems for which solutions have to be found in order to make an incremental version possible.

- How can the sample be split, or alternatively, how can several sequences of invocations of the examples routine return sentences of increasing complexity?
- How can the results obtained from the exploration of one substitution matrix be utilised for the exploration of the next?

The next paragraph deals with the first problem and in the paragraphs 5.3.3 and 5.3.4 I present three approaches to the second problem.

5.3.2 Structuring the sample space

It is helpful to understand the problem of an incremental version in terms of the algorithm presented in section 3.1. The improvements presented before affect the second stage of the algorithm in which the completeness result is determined. The proposed version is a modification such that in a loop the first stage is used to explore a somehow limited part of the entire universe of discourse for which the second stage provides the completeness results. A new, final stage combines the local completeness results into a larger result. For the domain of functions this would mean that for each iteration the first stage selects only values from a certain interval of the range, probably starting with intervals around certain interesting values such as 0, 1, π etc. Note that this means that knowledge about the domain is used to 'override' the probability distribution. The final stage combines the local results for several of these intervals to determine the result for the complete range, probably without having to consider each possible interval. It is also conceivable that the results for earlier intervals influence the selection of later intervals.

Trying to transfer this approach to the Emile algorithm makes it necessary to incorporate some knowledge about the target language that can be used to partition the sample space according to the complexity of the sentences. As the measure for complexity which is used throughout Adriaans' work, the Kolmogorov complexity, is not constructive it cannot be used for this purpose. Regarding other possible criteria it might seem that there is a significant correlation between the complexity and length of a sentence but this is generally not true for most languages and it is easy to provide an arbitrary amount of counterexamples. Also, any ambitious, linguistically motivated approach to determine the complexity of a sentence will take its internal structure into account but if a means is available to determine the structure of sentences, an application of the Emile algorithm is probably not necessary any more. Therefore, one can conclude that considering the structure of the sentence to determine its complexity is not a promising approach in this case.

New possibilities arise if information contained in the next smaller basic unit, the word, is considered. Often, complex constructions are introduced by certain function words. In English, for example, relative clauses are introduced by "that," "who" or "which" and coordination of phrases is usually achieved by conjunctions such as "or" and "and." A part-of-speech tagger[3] could be used to identify such word classes and sentences containing certain words could be grouped into batches. It is an open question how a ranking of the word classes regarding their effect on the complexity of a sentence could be established, and how this in turn could be used to determine the proper batch for a sentence containing several words belonging to one of these classes. Part-of-speech taggers also identify inflections with high reliability and based on that provide type information about each word. It would therefore be possible to consider, for example, only present tense verbs or singular nouns in the first batches. This approach seems promising for certain cases but it must be noted, that it makes a preprocessing stage necessary which requires further interaction. Also, part-of-speech taggers are not available for all languages and

3 See paragraph 1.2.4.

certainly not for special subsets of languages, such as the 'Call' database, which exhibits certain consistent idiosyncrasies which have to be considered.

Other criteria with which the sample space can be structured might exist for certain areas of application but I suspect that they depend on the particular area and cannot be generalised. This means that not only a preprocessing stage is necessary before the Emile algorithm can be used, but also that this ad hoc criterion has to be identified.

5.3.3 Utilising types and rules from previous runs

Provided that a satisfactory solution to split the sample is available, the next problem is to find a way in which the knowledge can be transferred between various runs of the algorithm. The result of one run of the algorithm is a grammar expressed in context-free rules. Encoded in this grammar are classes, or types, of words and sequences of words, as well as schemas stating how these can be combined to form a sentence. There are various strategies to utilise this knowledge to reduce the size of the substitution matrix of the next run. Note that these reductions apply to the smaller substitution matrices which means that the overall number of questions would be reduced even further.

Similar to the improvements presented in this chapter the type information could be used to substitute all members of a type by the type's representative in all sentences in the current batch. This would probably result in fewer sentences as several sentences can be equal after the replacements. Furthermore, given the current implementation the sentences would be shorter as the representative of a type is the shortest expression of that type. Both would lead to a smaller substitution matrix because its size is $O(s^2n^4)$ with s being the number of sentences in the sample and n the length of the longest sentence.

Another possibility would be to parse the sentences in the new batch as far as possible using the rules from previous runs. When creating the first order explosion all those basic rules in which the slash operators cross a bracketing established by a partial parse could be discarded. Considering an extended John-loves-Mary example a first run might have established types such as {the girl, a girl, …} and {the flower, the telescope, …} on the basis of some very simple sentences with intransitive verbs, nouns and determiners only.[4] If in a second batch prepositions are included, expressions of these types can be parsed in the corresponding example sentences.

(1) [The girl] with [the flower] walks

The unmodified first order explosion would create 21 basic rules from this example sentence including the following:

(2) S / with [the flower] walks $\rightarrow$ [the girl]
(3) [The \ S / [the flower] walks $\rightarrow$ girl] with
(4) [The girl] \ S / flower] walks $\rightarrow$ with [the

The modified version would discard rules (3) and (4), generating only 14 basic rules in this example. The main problem with this approach is the structural completeness of the Lambek

4 It is assumed that the language contains a verb such as "disappear" that has a single slot with an ITEM θ-role.

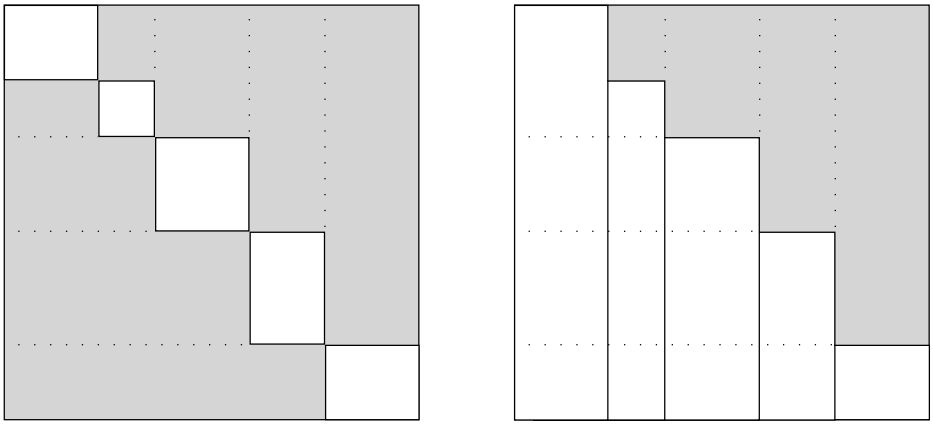


Figure 65. Ensuring discrimination of expressions in an incremental version.

calculus. Recalling previous results concerning the redundancy of the ruleset it is likely that the following partial parse would also be possible:

(5) [The girl] with [the [flower walks]

This effect worsens as sentences get longer with the result that almost no basic rules can be generated. Inverting the condition such that a basic rule is kept if a partial parse exists that does not make the slash operators enclose a bracket results in almost no rules being discarded.

There is a serious problem inherent in both approaches presented so far which is similar to the problem why contexts cannot be removed when an expression type is found.[5] An expression type is formed from expressions that can be inserted into the same set of contexts without altering grammaticality. In an incremental version the expressions are only tested in a fraction of all contexts, only those that are formed by the sentences in the current batch of example sentences, and it might be the case that two expressions are assigned to the same type because a discriminating context cannot be formed from the current batch. Implementing the proposal for forming batches on the basis of word types, i.e. their part-of-speech as determined in a preprocessing stage, the first batch might not contain prepositions and therefore the algorithm would assign "telescope" and "flower" to the same type because there is no context/expression pair forming a sentence with a verb having an instrument θ-role slot. If in a later batch prepositions are included, a context such as "Mary sees Mary with the flower with the \ *S*" or "Mary sees Mary with the telescope with the \ *S*" will occur, depending on which of the two words happened to become representative of the respective type. In both cases an inaccurate conclusion will be derived from the corresponding insertion. Either all of the members of the type are considered to fit into an instrument slot, or none are. The distinction between instruments and non-instruments is lost because no discriminating context was present in the sample in which the words occurred first.

This problem can be avoided if instead of 'accepting' the types from the previous run, all expressions belonging to a type are tested in all new contexts which ensures that a type can be split if necessary. Note, however, that this means that almost no knowledge is transferred between the runs. In the example "flower" and "telescope" might be assigned to the same type

5 See page 80.

in one run, but in the run containing the discriminating context the type would be split. The problem is indeed solved but at a high cost. As illustrated in figure 65 the expressions from the first batch are tested against all contexts from all batches, the expressions from the second batch are tested again all contexts from all but the first batch etc. Furthermore, occurrences of an expression cannot be replaced by the representative of its type anymore because the type might have to be split later. It therefore seems doubtful whether the number of questions will be lower than in the improvements presented before. It should also be noted that it can also happen that a discriminating context for two expressions occurs in an earlier batch. These cases are rare but they do distort the overall result. I would conclude that given these restrictions and the additional cost of a preprocessing stage the improvements discussed in this paragraph are by far less promising than the ones presented in sections 5.1 and 5.2.

5.3.4 Answering oracle questions with a previously learned grammar

It might seem intuitive to use the rules from previous runs to answer questions to the oracle. This approach would not only need a solution for the problem of transferring types between the runs, it is also doomed for another reason. The rules from previous runs cannot cover a larger subset of the target language than the subset considered in the current batch. This means that only for sentences that can be parsed, a positive answer can be returned instead of querying the oracle. For all sentences that cannot be parsed with the rules of the previous run it would still be necessary to query the oracle. Recalling the fact that usually more than 99% of the answers from the oracle are negative this means that using this approach only a few questions in the other 1% can be avoided.

5.4 Conclusions and future work

5.4.1 Estimating type memberships

Considering the severe problems regarding an incremental version of the Emile algorithm it seems more promising to investigate further strategies to remove rows and columns from the substitution matrix while exploring it. However, to my knowledge there are no further optimizations other than the ones described in sections 5.1 and 5.2 that do not in some way rely on statistics and/or heuristics. Given the fact that the Emile algorithm is located in the pac-learning framework it seems plausible to investigate how a parameter can be introduced to the algorithm such that in an extreme setting of the parameter a completely correct result is achieved and when modifying the parameter towards the opposite extreme, less and less correct grammars are learned.

The main appeal of an incremental version lies in the fact that is only necessary to check an expression in a fraction of the contexts in order to determine its type. However, this idea can be transferred to a non-incremental version and I will outline a simple solution in the remainder of this paragraph and present a more ambitious approach in the next paragraph.

When the algorithm determines the type of an expression in the current version, it first checks the validity of the expression in all contexts and then compares the set of contexts to the ones of existing types. This could also be incrementalised. After finding several contexts in which the expression is valid, the algorithm could compare this set of contexts to the sets of contexts of all

existing expression types. If the current set is a large enough subset of a set of a type then the expression could be assigned to that type. The required size of the subset would most likely depend on the number of contexts that have been checked, the size of the full set and a variable parameter. The parameter influences the learning process in a way which is common to algorithms in the pac-learning framework: If the value is large enough, the learning result will be perfect but running time (or oracle interaction) will be high, if the value is small then running time will be low but the probability that the learning result is correct and/or the quality of the result will be lower.

One point deserves further consideration. If the current set of contexts is a large enough subset of the sets of more than one expression type this indicates that the corresponding types belong to an existing type hierarchy. In this case one could assume that the expression belongs to one of the types and in order to find the proper subtype of the expression it should be sufficient to insert the expression into those contexts which do not belong to all of the sets.

5.4.2 A measure for the discriminatory power of contexts

In paragraph 5.1.1 I outlined the idea to remove all but one of the contexts from a set in which expressions of a newly found type are valid. I stated that this is not reasonable because usually no bipartite matching exists between expressions and contexts. Or, in other words, if a supertype is found and the contexts are removed, all chances to find a subtype are thwarted. However, a fraction of the contexts in which the expressions are checked would be sufficient to determine the general type and distinguish all the subtypes. Of course, one could randomly discard a certain percentage of the contexts, which is basically what the idea described in the previous paragraph does, and hope that enough contexts to discriminate the subtypes are retained. Another, more elaborate procedure is to introduce a metric according to which the discriminative power of two contexts regarding expression subtypes can be compared and check the context in a sequence starting with the most discriminative contexts towards contexts that distinguish between subtypes.

Not unlike the general idea of Baysian text classifiers a vector is associated with each context. The dimensionality is determined by the number of expression types formed and does therefore grow while the algorithm is running. Each row contains a 1 if expressions of the corresponding type are valid in this context and a 0 otherwise. Before checking an expression the algorithm calculates the angle between all vectors and picks a number of contexts pointing into different regions of the vector space. When it finds a number of contexts in which the expression is valid it has determined a general direction and can continue checking contexts in the proximity of those vectors to determine the exact subtype of the expression. This procedure can be repeated several times. The algorithm stops if the set of contexts in which the current expression was found to be valid in is equal to the set of one of the types in a vector.

Conclusion

Despite the somewhat negative results of the last chapters I believe that Adriaans' model of cooperative language learning is a great step forward after Gold's model of identification in the limit. It provides a much better characterization of the complexity of language learning and a more realistic framework to evaluate approaches in this area. Moreover, I still trust the general assumptions that Adriaans employs to arrive at his learnability results concerning the task of learning a grammar for natural languages. Regarding the actual implementation of this model in form of the Emile algorithm, however, I would conclude that without extensive modifications the algorithm is not applicable to real-world problems. The improvements that I have considered retain the elegance and simplicity of the original algorithm and because of that are probably not far-reaching enough to achieve reasonable practical success. Also, the appeal of categorial grammar theory is often seen in the combination of syntactical and semantic properties of languages but this aspect is clearly neglected in the Emile algorithm. It does, however, receive appropriate attention in Adriaans' original work.

In order to provide a basis for future work I will summarise the problems encountered in a practical application of the Emile algorithm.

⇒ The characterization of the examples routine is not constructive.

This makes it impossible to estimate the sample size in advance which would be highly desirable. The approach employed in section 3.4, namely to experiment with various sample sizes and, if in doubt, choose a larger sample is not practical in applications where a person answers the questions. In such a case the obvious conflict between the interest to keep the sample as small as possible to limit the number of questions and the need to choose a large enough sample to achieve learnability cannot be resolved in a satisfactory manner.

Furthermore, the characterization of the examples routine with a probability distribution is not strict enough for incremental versions. If the algorithm can only assume that more complex sentences are returned with an (exponentially) smaller probability but has no means to estimate the complexity of a given sentence it cannot postpone more complex sentences. This problem was discussed in paragraph 5.3.2.

⇒ The algorithm is not efficient enough.

In complexity theory an algorithm is considered efficient if its running-time and space are polynomially bounded in the length of the input and the Emile algorithm clearly satisfies this criterion. However, the size of the substitution matrix is still by far too large to be answered by a person. In fact, for any real-world application it is even too large to be processed automatically by a parser, given the CPU power and memory constraints of contemporary workstations. See paragraphs 3.4.5 and 3.5.2.

⇒ The standards expected from the oracle are too high.

While at a first glance it might seem simple, even trivial for a person to act as an oracle and determine whether a given sentence is correct, this is one of the main problems of the algo-

rithm. For a start, all questions, the number of which will be at least in the order of tenths of thousands, must be answered consistently. The improvements presented in sections 5.1 and 5.2 alleviate this problem because they infer answers for similar questions and thus avoid to ask questions that could be answered in a clearly inconsistent way. A general technique to handle noise in oracle routines that Adriaans mentions, namely to ask each question several times and choose the prevailing answer, is obviously not plausible as it would increase the number of questions considerably.

Moreover, the idea of the oracle routine builds on the intuitive linguistic ability of the (native) speaker to determine whether or not a sentence is correct. This is problematic because it is generally hard to distinguish between syntactical incorrectness, 'wrong' combinations of words such as "water sleeps" and sentences that carry a meaning which is false, such as "Flowers are animals." The latter problem can be overcome with some concentration but it is very hard to tackle the former. One could argue that such answers should be allowed but under these circumstances one has to bear in mind that the underlying model must be expressible in a shallow CFG. See paragraph 3.2.3. The intuitive notion of correctness is also problematic insofar as this implies that nothing but the complete language can be learned. I believe that it is impossible for any speaker to determine for not well delineated subsets of a language, such as the one from the 'Call' database in paragraph 3.4.5, whether a sentence belongs to it or not. The speakers will always more or less answer the oracle questions according to their native language.

Finally, the level of interaction with a person as an oracle is that of performance in Chomskian terms. This means that the model will not describe language according to the customary linguistic understanding, i.e. competence, but will also account for performance limitations as explained in paragraph 3.5.2. Whether this is acceptable remains to be seen.

⇒ **The ruleset is too redundant.**

As described throughout section 3.4 the set of rules learnt by the Emile algorithm is highly redundant. Part of the redundancy stems from the insertion of a single expression type into the sequence of words representing the respective context. This redundancy can be removed to some extent with the algorithms described in sections 4.1 and 4.2. The generation of another type of redundancy, which I called structural redundancy because it is the result of the structural completeness of the Lambek calculus, seems to be an intrinsic property of the algorithm. It is possible that a sufficient amount of the redundant rules can be removed by a post-processing stage, which might implement the ideas outlined in paragraph 4.4.1, or that the redundancy can be tolerated in the set of rules if measures, such as the one presented in paragraph 4.4.3, that provide a consistent ranking of the parse trees are found. In my opinion this problem should be the foremost target of any future work, because the algorithm might still be useful for other purposes such as transforming a hand-written grammar into a more efficient form, as described in paragraph 3.4.5, even if the number of questions to the oracle cannot be reduced considerably. If, however, no satisfactory solution can be found to reduce the redundancy of the rule set then the learnt grammar is of little value for any purpose.

Given the massive problems described above one could ask how likely it is that alternative solutions can be found at all. Regarding grammar learning tasks in which a hand-written core grammar should be extended I would assume that algorithms such as the one that James Cussens et al. described[1] would be more suitable. One of the main advantages is that the number of oracle questions is considerably smaller because seemingly more complicated questions are asked.

However, as outlined above, the possibility to directly evaluate the rules might even be preferable. For large amounts of data in which some simple structure has to been found, the Sequitur algorithm[2] seems promising. The authors show that it can learn a grammar describing its input on a structural level from positive examples in linear time. It should be noted that the approach does not generalise too well and therefore the grammar will often not be applicable to unknown input. For certain areas of applications, however, this is no disadvantage. Recalling the theoretical value of Adriaans' framework, it would be interesting to attempt an evaluation of these algorithms in this framework.

1 See paragraph 2.2.4 on page 23.

2 [Nevill-Manning and Witten, 1997]

Appendix

A Overview of the Experiments

The following tables summarize the improvements achieved with regard to experiment 3. As the grammars are prohibilitvely large, those for experiment 2 were included instead.

Experiment / Sample	3 / #2			3 / #6		
Algor. from section	3.3	5.1	5.2	3.3	5.1	5.2
questions	143 079	40 303	32 875	318 043	53 044	44 208
...relative to matrix	77.8%	21.9%	17.9%	79.1%	13.2%	11.0%
positive answers	0.7%	1.0%	0.9%	0.6%	1.0%	1.0%
types	131	129	126	137	137	131
rules	1 165	840	855	1 495	1 010	1 000
...from contexts	872	682	698	1 089	842	833
undergeneration	0.06	0	0	0	0.01	0.01
overgeneration	0	0	0.61	0	0	0.60
average ambiguity	20.4	12.2	8.7	31.2	15.3	10.0

Table 1. Results of the improvements presented in section 5.1 and 5.2.

Algor. from section	3.3			5.1			5.2		
Restructuring	–	U+S	HUS	–	U+S	HUS	–	U+S	HUS
rules	1495	882	580	1010	571	514	1000	576	501
...from contexts	1089	556	208	842	403	223	833	408	231
...shortened		727	95		430	61		433	96
average ambiguity	31.2	16.8		15.3	8.8		10.0	5.9	

Table 2. Results for combinations of various improvements for experiment 3, sample #6

Algorithm	Parses
GULP grammar	[[john] [meets [[the man] [with [the telescope]]]]]
original	[[john meets the man] with the telescope] [[john meets the] man with the telescope] [[john meets] [the man] with the telescope] [[john meets] the man with the telescope] [john [meets [the man] with the] telescope] [john [meets [the man] with] the telescope] [john [meets [the man]] with the telescope] [john [meets the man] with the telescope] [john [meets the] man with the telescope] [john meets [[the man] with the telescope]] [john meets [the man with the telescope]] [john meets the [man with the telescope]] [john meets the man with the telescope]
original with restructuring U and S	[[john [meets the] man] [with [the telescope]]] [[john [meets the]] man [with [the telescope]]] [[john meets] [the man] [with [the telescope]]] [john [meets [[the man] [with the]]] telescope] [john [meets [[the man] with]] [the telescope]] [john [meets [the man]] [with [the telescope]]] [john [meets the] [man [with [the telescope]]]] [john [meets the] man [with [the telescope]]] [john meets [[the man] [with [the telescope]]]]
from section 5.1	[[john meets [the man]] with the telescope] [[john meets the] man with the telescope] [[john meets] [the man] with the telescope] [[john meets] the man with the telescope] [john [meets [the man]] with the telescope] [john [meets the] man with the telescope] [john meets [[the man] with the telescope]] [john meets [[the man] with the] telescope] [john meets the [man with the telescope]] [john meets the man with the telescope]

Table 3. Sets of parse-trees for the sentence "John meets the man with the telescope" generated by the grammar learnt from sample #6 in experiment 3.

Algorithm	Parses
from section 5.2	[[john meets [the man]] with the telescope] [[john meets the] man with the telescope] [[john meets] [the man] with the telescope] [[john meets] the man with the telescope] [john [meets [the man]] with the telescope] [john [meets the] man with the telescope] [john meets [[the man] with the telescope]] [john meets [[the man] with the] telescope] [john meets the [man with the telescope]] [john meets the man with the telescope]
from section 5.2 with restructuring U and S	[[[john meets] [the man]] [with [the telescope]]] [[john [meets the]] [man [with [the telescope]]]] [[john [meets the]] man [with [the telescope]]] [[john meets] [[the man] [with [the telescope]]]] [[john meets] [[the man] [with the]] telescope] [john [meets [the man]] [with [the telescope]]]

Table 3. Sets of parse-trees for the sentence "John meets the man with the telescope" generated by the grammar learnt from sample #6 in experiment 3.

Algor.	original		from section 5.2	
Restr.	none	U+S	none	U+S
	$S \to C\,B\,A$	$S \to C\,B\,F\,A$	$S \to C\,B\,A$	$S \to C\,D\,E$
	$S \to C\,B\,C\,B\,A$	$S \to C\,B\,G$	$S \to C\,B\,C\,B\,A$	$S \to F\,A$
	$S \to C\,B\,C\,B\,G$	$S \to C\,D\,B\,A$	$S \to C\,B\,C\,E$	$S \to F\,F\,A$
	$S \to C\,B\,C\,E$	$S \to C\,D\,B\,G$	$S \to C\,B\,F\,A$	$S \to F\,G$
	$S \to C\,B\,F\,A$	$S \to C\,D\,E$	$S \to C\,B\,G$	$S \to G$
	$S \to C\,B\,G$	$S \to C\,E$	$S \to C\,D\,B\,A$	
	$S \to C\,D\,B\,A$	$S \to F\,A$	$S \to C\,E$	
	$S \to C\,E$	$S \to F\,C\,B\,A$	$S \to F\,A$	
	$S \to F\,A$	$S \to G$	$S \to F\,C\,B\,A$	
	$S \to F\,C\,B\,A$		$S \to G$	
	$S \to G$			
	$B \to B\,C\,B$	$B \to D\,B$	$B \to B\,C\,B$	$B \to B\,F$
	$C \to C\,B\,C$	$C \to C\,D$	$C \to C\,B\,C$	$C \to C\,D$
	$D \to B\,C$	$D \to B\,C$	$D \to B\,C$	$D \to B\,C$
	$E \to B\,A$	$E \to B\,A$	$E \to B\,A$	$E \to B\,A$
	$E \to B\,C\,B\,A$	$E \to D\,B\,A$	$F \to C\,B$	$F \to C\,B$
	$F \to C\,B\,C\,B$	$F \to C\,B$	$G \to C\,B\,A$	$G \to C\,E$
	$F \to C\,B$	$F \to C\,D\,B$		
	$G \to C\,B\,A$	$G \to C\,B\,A$		
	$G \to C\,B\,C\,B\,A$	$G \to C\,D\,B\,A$		
	$A \to$ c	$A \to$ c	$A \to$ c	$A \to$ c
	$B \to$ b	$B \to$ b	$B \to$ b	$B \to$ b
	$C \to$ a	$C \to$ a	$C \to$ a	$C \to$ a

Table 4. Grammars learnt for the language in experiment 2.

Appendix

B Grammar for Experiment 3

```
% ---------------------------------------------------------------
% SCHEMA
% ---------------------------------------------------------------

% verbtype: intr, itra, tr, tra
g_schema(vtype:X, [X,_,_,_,_,_,_]).

% verbframe:
g_schema(subth:X, [_,X,_,_,_,_,_]).
g_schema(objth:X, [_,_,X,_,_,_,_]).
g_schema(adjth:X, [_,_,_,X,_,_,_]).

% theta roles: agent, instr, item
g_schema(theta:X, [_,_,_,_,_,X,_]).

% nountype: proper, norm
g_schema(ntype:X, [_,_,_,_,_,_,X]).

% ---------------------------------------------------------------
% RULES
% ---------------------------------------------------------------

s(s(NP, VP))
--> np(NP, theta:S), vp(VP, theta:S).

vp(vp(V), theta:S)
--> verb(V, vtype:intr..subth:S).
vp(vp(V), theta:S)
--> verb(V, vtype:itra..subth:S).
vp(vp(V, PP), theta:S)
--> verb(V, vtype:itra..subth:S..adjth:A), pp(PP, theta:A).
vp(vp(V, NP), theta:S)
--> verb(V, vtype:tr..subth:S..objth:O), np(NP, theta:O).
vp(vp(V, NP), theta:S)
--> verb(V, vtype:tra..subth:S..objth:O), np(NP, theta:O).
vp(vp(V, NP, PP), theta:S)
--> verb(V, vtype:tra..subth:S..objth:O..adjth:A), np(NP, theta:O),
    pp(PP, theta:A).

pp(pp(P, NP), theta:TH)
--> prep(P, theta:TH), np(NP, theta:TH).

np(NP, theta:X)
--> snp(NP, theta:X).
np(np(NP, PP), theta:agent)
```

```
--> snp(NP, theta:agent), pp(PP, theta:instr).
np(np(NP, PP), theta:agent)
--> snp(NP, theta:agent), pp(PP, theta:item).

snp(np(N), theta:TH)
--> noun(N, ntype:proper..theta:TH).
snp(np(D, N), theta:TH)
--> det(D), noun(N, ntype:norm..theta:TH).

% ---------------------------------------------------------------
% LEXICON
% ---------------------------------------------------------------

verb(loves,   vtype:tr..subth:agent)               --> [loves].
verb(finds,   vtype:tr..subth:agent)               --> [finds].
verb(invites, vtype:tr..subth:agent..objth:agent)  --> [invites].
verb(meets,   vtype:tr..subth:agent..objth:agent)  --> [meets].
verb(sees,    vtype:tra..subth:agent..adjth:instr) --> [sees].
verb(walks,   vtype:itra..subth:agent..adjth:agent) --> [walks].

noun(mary,      ntype:proper..theta:agent) --> [mary].
noun(john,      ntype:proper..theta:agent) --> [john].
noun(girl,      ntype:norm..theta:agent)   --> [girl].
noun(man,       ntype:norm..theta:agent)   --> [man].
noun(flower,    ntype:norm..theta:item)    --> [flower].
noun(telescope, ntype:norm..theta:instr)   --> [telescope].

det(the) --> [the].
det(a)   --> [a].

prep(with, theta:agent) --> [with].
prep(with, theta:instr) --> [with].
prep(with, theta:item)  --> [with].
```

Bibliography

N. Abe, 1988. “Learnability and locality of formal grammars.” In *Proceedings of the 26th Annual meeting of the Association of computational linguistics*.

Pieter W. Adriaans, 1992. *Language Learning from a Categorial Perspective*. PhD thesis. Universiteit van Amsterdam.

Pieter W. Adriaans and Arno J. Knobbe, 1996. “EMILE: Learning Context-free Grammars from Examples.” Proposed Paper for IML-96.

Kazimierz Ajdukiewicz, 1935. “Die syntaktische Konnexität.” In *Studia Philosphica*, **1**: 1– 27.

Dana Angluin, 1987. “Learning regular sets from queries and counter-examples.” In *Information and Computation*, **75**: 87–106.

Dana Angluin, 1987b. *Learning k-bounded context-free grammars*. Technical Report. Yale University, Department of Computing Science.

Ken Arnold and James Gosling, 1996. *The Java programming language*. (2nd Ed.) Java Series. Reading, MA: Addison Wesley.

Richard Beckwith, George A. Miller and Randee Tengi, 1993. *Design and Implementation of the WordNet Lexical Database and Searching Software*. Technical Report. Princeton University.

Georg Berg, 1992. “A Connectionist Parser with Recursive Sentence Structure and Lexical Disambiguation.” In *Proceedings: 10th National Conference in Artificial Intelligence AAAI-92*. AAAI. pp. 32–37.

Margaret A. Boden (editor), 1990. *The Philosophy of Artificial Intelligence*. Oxford Readings in Philosophy. Oxford: Oxford University Press.

Ted Briscoe and John Carroll, 1993. “Generalized Probabilistic LR Parsing of Natural Language (Corpora) with Unification-Based Grammars.” In *Computational Linguistics*, **19**(1): 24–59.

W. Buszkowski, W. Marciszewski and J. van Benthem (editors), 1988. *Categorial Grammar.* Amsterdam: Benjamins.

W. Buszkowski, 1988. “Generative Power of categorial Grammars.” In *Categorial Grammars and Natural Language Structures*. R. T. Oehrle, E. Bach and D. Wheeler, editors. Dordrecht: D. Reidel Publishing Company.

Eugene Charniak, 1995. *Statistical Language Learning*. Cambridge, MA: MIT Press.

Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, 1992. *Introduction to algorithms*. The MIT electrical engineering and computer science series. Cambridge, MA: MIT Press.

Michael A. Covington, 1994. *Natural language processing for Prolog programmers*. Englewood Cliffs, NJ: Prentice Hall.

Brad J. Cox and Andrew J. Novobilski, 1991. *Object-Oriented Programming. An evolutionary approach*. (2nd Ed.) Reading, MA: Addison Wesley.

James Cussens, David Page, Stephen Muggleton and Ashwin Srinivasan, 1997. "Using Inductive Logic Programming for Natural Language Processing." In *9th European Conference on ML: Workshop Notes on Empirical Language Processing Tasks*. Walter Daelemans, Ton Weisters and Antal van der Bosch, editors. pp. 25–34.

Jay Earley, 1970. "An efficient Context-free parsing algorithm." In *Communications of the ACM*, **6**(8): 94–102.

J. Fujisaki *et al*, 1989. "A probabilistic method for sentence disambiguation." In *Proceedings of the 1st Internation Workshop on Parsing Technologies*. pp. 105–114.

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Reading, MA: Addison-Wesley.

Gerald Gazdar, 1988. "Applicability of Indexed Grammars to Natural Languages." In *Natural Language Parsing and Linguistic Theories*. U. Reyle and C. Rohrer, editors. Dordrecht: D. Reidel Publishing Company. pp. 64–94.

Talmy Givón, 1993. *English grammar. A function based introduction*. (Vols. I and II). Amsterdam: Benjamins.

Talmy Givón, 1995. *Functionalism and grammar.* Amsterdam: Benjamins.

E. Mark Gold, 1967. "Language Identification in the Limit." In *Information and Control*, **10**: 447–474.

Liliane Haegeman, 1991. *Introduction to government and binding theory.* Oxford: Blackwell.

Zellig Harris, 1951. *Methods in Structural Linguistics*. Technical Report. University of Chicago Press.

Zellig Harris, 1964. "Distributional Structure." In *The Structure of Language*. J. A. Fodor and J. J. Katz, editors. New York: Prentice Hall.

C. F. Hockett (editor), 1958. *A Course in Modern Linguistics*. New York, NY. Macmillian.

Markus Hölscher, 1997. *Informationsextraktion aus Freitext-Einträgen einer Datenbank*. Thesis. Universität Dortmund.

Donald E. Knuth, James H. Morris and Vaughan R. Pratt, 1977. "Fast Pattern Matching in Strings." In *SIAM Journal on Computing*, **6**(2): 323–350.

Sydney M. Lamb, 1961. “On the mechanization of syntactic analysis.” In *1961 Conference on Machine Translation of Languages and Applied Language Analysis*. London: Her Majesty's Stationery Office. pp. 674–685.

Marc M. Lankhorst, 1994. “A Genetic Algorithm for the Induction of Context-free Grammars.” In *CLIN IV. Papers from the Fourth CLIN Meeting*. G. Bouma and G. van Noord, editors. pp. 87–100.

Ming Li and Paul M.B Vitányi, 1990. “Kolmogorov complexity and its applications.” In *Handbook of Theoretical Computer Science, Vol. 1*. J. van Leeuwen, editor. Amsterdam: Elsevier Science Publishers. pp. 188–254.

Ming Li and Paul M.B. Vitányi, 1991. “Learning simple concepts under simple distributions.” In *SIAM Journal on Computing*, **29**(5): 991–935.

S. McCall (editor), 1967. *Polish Logic 1920 – 1939*. Oxford, Clarendon Press.

Mary McGee Wood, 1993. *Categorial Grammars*. London: Routledge.

George A. Miller *et al*., 1993. *Introduction to WordNet: An On-Line Lexical Database*. Technical Report. Princeton University.

Marvin Minsky, 1988. *The Society of Mind*. Simon and Schuster.

Katharina Morik, 1995. “Maschinelles Lernen.” In *Einführung in die Künstliche Intelligenz*. Günther Görtz, editor. Bonn: Addison-Wesley. pp. 243–297.

Craig G. Nevill-Manning and Ian H. Witten, 1997. “Identifying Hierarchical Structure in Sequences: A linear-time algorithm.” In *Journal of Artificial Intelligence Research*, **7**: 67–82.

R. T. Oehrle, E. Bach and D. Wheeler (editors), 1988. *Categorial Grammars and Natural Language Structures*. Dordrecht: D. Reidel Publishing Company.

Mati Pentus, 1993. “Lambek Grammars are Context Free.” In *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science*. Los Alamitos, CA: IEEE Computer Society Press. pp. 429–433.

Fernando C. N. Pereira and David H. D. Warren, 1980. “Definite Clause Grammars for Language Analysis. A Survey of the Formalism and a Comparison with Augmented Transition Networks.” In *Artificial Intelligence*, **13**: 231–278.

Jorma Rissanen, 1978. “Modeling by shortest data description.” In *Automatica*, **14**: 465–471.

Yasubumi Sakakibara, 1988. “Learning Context-Free Grammars from Structural Data in Polynomial Time.” In *Proceedings of the 1988 Workshop on Computational Learning Theory.* David Haussler and Pitt Leonard, editors. San Mateo, CA: Morgan Kauffman. pp. 330–344.

Stuart M. Shieber *et al*., 1983. “The Formalism and Implementation of PATR-II.” In *Research on Interactive Acquisition and Use of Knowledge*. B. Grosz and M. E. Stickel, editors. Menlo Park, CA: SRI International. pp. 39–79.

Stuart M. Shieber, 1985. “Evidence against the Context-freeness of Natural Language.” In *Linguistics and Philosophy,* **8**: 333–343.

Robert F. Simmons and Yeong-Ho Yu, 1992. “The Acquisition and Use of Context-Dependent Grammars for English.” In *Computational Linguistics*, **18**(4): 391–418.

Edgar Sommer, 1996. *Theory Restructuring. A Perspective on Design & Maintenance of Knowledge Based Systems*. PhD thesis. Universität Dortmund.

Jon Spencer, 1996. *Now I Got Worry.* London: Mute Records.

Masaru Tomita and See-Kiong Ng, 1991. “The Generalized LR Parsing Algorithm.” In *Generalized LR Parsing*. Masaru Tomita, editor. Boston, MA: Kluwer Academic Publishers. pp. 1–16.

Masaru Tomita, 1986. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Boston, MA: Kluwer Academic Publishers.

D. S. Touretzky (editor), 1991. *Machine Learning, Special Issue on Connectionst Approaches to Language Learning*. Boston, MA: Kluwer Academic Publishers.

Allan M. Turing, 1950. “Computing Machinery and Intelligence.” In *Mind*, **59**: 433–460.

L. G. Valiant, 1984. “A Theory of the Learnable.” In *Communications of the ACM*, **27**(11): 1134–1142.

Eric Wehrli, 1988. “Parsing with a GB-Grammar.” In *Natural Language Parsing and Linguistic Theories*. U. Reyle and C. Rohrer, editors. Dordrecht: D. Reidel Publishing Company. pp. 177–201.

T. Yokomori, 1989. “Learning Context-Free Languages Efficiently: A Report on Recent Results in Japan.” In *Analogical and Inductive Inference: Proceedings of the International Workshop AII'89*. K. P. Jantke, editor. Berlin, Heidelberg: Springer. pp. 104–123.

D.H. Younger, 1967. “Recognition and Parsing of Context-free Languages in time n^3.” In *Information and Control*, **10**(2): 189–208.