Semantic Syntax: Evaluation by Implementation
Cover illustration: film still taken from *Father and Daughter* by Michael Dudok de Wit, 2000 (with kind permission from Leopold).

Published by
LOT
Trans 10
3512 JK Utrecht
The Netherlands

ISBN 90-76864-12-8

© 2001 by Lisanne Teunissen. All rights reserved.
Printed in The Netherlands.
Semantic Syntax:
Evaluation by Implementation

Een wetenschappelijke proeve
op het gebied der Letteren

door
Lisanne Maria Teunissen
## Contents

<table>
<thead>
<tr>
<th>ACKNOWLEDGEMENTS</th>
<th>IX</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER 1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 PREAMBLE</td>
<td>1</td>
</tr>
<tr>
<td>1.2 RESEARCH OBJECT</td>
<td>2</td>
</tr>
<tr>
<td>1.2.1 SEMANTIC SYNTAX</td>
<td>2</td>
</tr>
<tr>
<td>1.2.2 RESEARCH QUESTIONS</td>
<td>6</td>
</tr>
<tr>
<td>1.3 METHODOLOGY</td>
<td>7</td>
</tr>
<tr>
<td>1.3.1 TECHNOLOGICALISTIC</td>
<td>7</td>
</tr>
<tr>
<td>1.3.2 DELIMITATIONS</td>
<td>9</td>
</tr>
<tr>
<td>1.4 RELATED RESEARCH</td>
<td>12</td>
</tr>
<tr>
<td>1.4.1 SEMANTIC SYNTAX</td>
<td>12</td>
</tr>
<tr>
<td>1.4.2 TECHNOLOGICALISTIC</td>
<td>15</td>
</tr>
<tr>
<td>1.5 CONCLUSION</td>
<td>16</td>
</tr>
<tr>
<td>CHAPTER 2 NATURAL LANGUAGE GENERATION</td>
<td>19</td>
</tr>
<tr>
<td>2.1 INTRODUCTION</td>
<td>19</td>
</tr>
<tr>
<td>2.2 ASPECTS OF NATURAL LANGUAGE GENERATION</td>
<td>20</td>
</tr>
<tr>
<td>2.2.1 PURPOSE</td>
<td>20</td>
</tr>
<tr>
<td>2.2.2 TASK</td>
<td>21</td>
</tr>
<tr>
<td>2.2.3 METHOD</td>
<td>21</td>
</tr>
<tr>
<td>2.2.4 TOOLS</td>
<td>22</td>
</tr>
<tr>
<td>2.3 TECHNOLOGICALISTIC APPROACH TO GENERATION</td>
<td>23</td>
</tr>
<tr>
<td>2.3.1 PURPOSE</td>
<td>23</td>
</tr>
<tr>
<td>2.3.2 TASK</td>
<td>24</td>
</tr>
<tr>
<td>2.3.3 METHOD</td>
<td>24</td>
</tr>
<tr>
<td>2.3.4 TOOLS</td>
<td>24</td>
</tr>
<tr>
<td>2.4 CONCLUSION</td>
<td>25</td>
</tr>
<tr>
<td>CHAPTER 3 FORMATION RULES AND LEXICON</td>
<td>29</td>
</tr>
<tr>
<td>3.1 INTRODUCTION</td>
<td>29</td>
</tr>
</tbody>
</table>
3.2 FORMATION RULES 30
3.2.1 THE TENSES 30
3.2.2 LEXICAL ARGUMENT FRAME 32
3.2.3 MODALITY AND FUTURE TENSE 35
3.2.4 MIDDLE AUXILIARIES 38
3.2.5 ADVERBIALS 41
3.2.6 TOPICALIZATION AND QUESTIONS 45
3.2.7 PROGRESSIVE 51
3.2.8 PASSIVE AND EXTERNAL DATIVE 52
3.2.9 NOUN PHRASES 58
3.2.10 QUANTIFIERS 61

3.3 THE LEXICON 65

3.4 CONCLUSION 68
3.4.1 DESCRIPTIVE ADEQUACY 68
3.4.2 CROSSLINGUISTIC ADEQUACY 72

CHAPTER 4 THE TRANSFORMATIONAL GRAMMAR FOR ENGLISH 75

4.1 INTRODUCTION 75

4.2 PRECYCLE 77
4.2.1 SA FILTER 78
4.2.2 AFFIX-TO-FEATURE-BUNDLE AND S-TYPE 80
4.2.3 NP-PRUNING 83
4.2.4 ARGUMENT FUNCTIONS 84
4.2.5 NEW SENTENCE UNIT AND REST OF SENTENCE UNIT 85
4.2.6 SPINE 87
4.2.7 LOOKUP RULE FEATURES 89
4.2.8 PERFECTIVE AUXILIARY 89

4.3 CYCLE 90
4.3.1 PARTICIPLE 93
4.3.2 LOWERING 94
4.3.3 TREE PRUNING 99
4.3.4 SURFACE CATEGORY ASSIGNMENT 103
4.3.5 SUBJECT RAISING 104
4.3.6 PARTICLE-INSERTION 106
4.3.7 SUBJECT DELETION 109
4.3.8 OBJECT INCORPORATION 112
4.3.9 EXTRAPOSITION 113
4.3.10 COPULA INSERTION 115
4.3.11 COMPLEMENTIZER INSERTION 118

4.4 POSTCYCLE 119
4.4.1 AUX-DELIMITATION 120
4.4.2 SIM-DELETION 122
4.4.3 ADVERB PLACEMENT 123
4.4.4 ATTRACTION 126
4.4.5 DO-SUPPORT 131
4.4.6 AFFIX HANDLING 133
4.4.7 MINCE 135

4.5 CONCLUSION 136

CHAPTER 5 THE TRANSFORMATIONAL GRAMMAR FOR DUTCH AND GERMAN 143
5.1 INTRODUCTION 143
5.2 GRAMMAR PARAMETRIZATION 143
  5.2.1 PREDICATE RAISING 144
  5.2.2 PARTICLE-PARTICIPLE 148
  5.2.3 V-FINAL 149
  5.2.4 END-CLUSTER-ARRANGEMENT 152
5.3 LEXICAL RULE PARAMETRIZATION 155
  5.3.1 LEXICON LOOKUP 156
  5.3.2 LEXICAL FILLERS 156
5.4 BRANCHING PARAMETRIZATION 158
  5.4.1 LEFT-BRANCHING GERMAN CLUSTERS 160
  5.4.2 DUTCH L-VERBS AND GERMAN R-VERBS 168
5.5 OTHER ADJUSTMENTS TO THE RULES 174
  5.5.1 ARGUMENT FUNCTIONS 174
  5.5.2 PERFECTIVE AUXILIARY 175
  5.5.3 PARTICIPLE 176
  5.5.4 LOWERING 178
  5.5.5 PARTICLE-INSERTION 179
  5.5.6 SIM-DELETION 180
5.6 CONCLUSION 181
  5.6.1 DESCRIPTIVE ADEQUACY 181
  5.6.2 CROSSLINGUISTIC ADEQUACY 182

CHAPTER 6 SEMANTIC SYNTAX PARSING 185
6.1 INTRODUCTION 185
6.2 THE SURFACE PARSER AMAZON 187
6.3 A BASIC SEMANTIC PARSER 189
  6.3.1 FROM SS TO SA 189
  6.3.2 THE TENSE SYSTEM 191
6.3.3 ARGUMENT STRUCTURE 195
6.3.4 TOPICALIZATION AND QUESTIONS 198

6.4 CONCLUSION 199

REFERENCES 203

APPENDIX: GRAMGEN 207
Acknowledgements

I would like to thank all the people who have helped me complete this thesis. First and foremost, my supervisors Pieter Seuren and Peter-Arno Coppen. Without their enthusiasm, their vast knowledge, and their stimulating support, I would never have been able to do the job – in fact, I would probably not even have started it. The interesting discussions we had always made me realize how fascinating language and linguistics really are.

Furthermore, a big thank-you goes to all my colleagues at the Department of General Linguistics and Dialectology, who provided me with an inspiring environment. Over the past few years, it was good to be able to share my coffee and lunch breaks as well as my ideas and frustrations with you. I would especially like to thank Henk Schotel, with whom I shared an office for a while – I enjoyed our conversations a lot! I think he was one of the very few colleagues who could actually understand what my research was all about. I should also mention the ProZA and POEZie groups, which always offered a good setting for putting things in perspective. And Marie-José, thank you for agreeing to assist me during the defence of my thesis even though the very next day you are to defend yours!

Finally, I would like to thank my friends and relatives for keeping me in touch with everyday life. My parents for encouraging me and for never failing to show great interest in my work, and Jan-Willem for being there, always.
Het fietsen deed mij goed, deels omdat het als beweging al zo sterk geassocieerd was met de lange fietstocht naar huis, deels vanwege de beweging zelf die je uitvoerde zonder hulp van anderen en welke je tegelijkertijd vooruit bracht en overeind hield.

[uit: Maarten 't Hart, Een vlucht regenwulpen, 1978]
CHAPTER 1  Introduction

1.1  Preamble

In 1996, Pieter Seuren published his *Semantic Syntax*, a book “meant not only for theoretical linguists but also for computational linguists” (Seuren 1996, p.xi). The book presents a syntactic theory named Semantic Syntax, developed by Seuren over the preceding three decades within the tradition of Generative Semantics. The theory is represented by means of rule systems for a number of different languages. The rules map semantic analyses of sentences onto syntactic surface structures (see section 1.2.1).

Seuren’s methodology in developing Semantic Syntax has been largely inductive. The starting point of his linguistic enterprise has always been formed by the facts of language, not by a priori principles:

“Formal a prioris are less helpful since language always turns out to be different from what one expects it to be, and more complex as well. [...] inductive delving into the ecological reality of language is necessary if one wants to find out about it.” (Seuren 1996, p.9)

The book shows that this methodology has certainly been fruitful, since many often notoriously puzzling phenomena of the languages in question are accounted for in a uniform manner. But the question arises as to the status of the a posteriori established generalizations.

“We know, of course, that in the vast majority of cases there is no way of proving formally that a particular description or analysis is the correct one. But we know equally well that analyses that keep being confirmed from often unexpected quarters and at the same time have a high degree of generality in terms of formal apparatus and coverage of facts do acquire, in a very real sense, a ring of truth. And if such analyses begin to be numerous one gets the feeling of being somehow on the right track.” (ibid. p.xiii)

Formal proof might well be out of reach, but one may ask whether the “ring of truth” and the “feeling of somehow being on the right track” cannot be put to some objective test. This in fact constitutes the central research question underlying the present thesis. What are the merits of Semantic Syntax on which it can be judged, and what are its flaws? Can both of these aspects be identified in a formal manner? These questions are not trivial. In our view, a sound evaluation of this syntactic theory is imperative before we can move on to a stage where:

“[...] interesting wider perspectives will no doubt open up for the philosophy of man and his mental life, for the vastness as well as the limitations of language, and also for practical applications in this new era where the computer is penetrating into so many spheres of life.”

(ibid. p.xiii-xiv)
In the remainder of this introductory chapter, I will first describe in greater detail the model of Semantic Syntax and the questions we will seek to answer with respect to it (section 1.2). Next, I will go into our technolinguistic methodology and the delimitation of our research area (section 1.3) and some related research in the areas of both Semantic Syntax and technolinguistics (section 1.4). Finally, I will recapitulate what has been said and clarify the outline of the remainder of this thesis (section 1.5).

1.2 Research object

1.2.1 Semantic Syntax

It is important to stress that the object of this research is not language, or some language in particular, but rather a theory about language, namely Semantic Syntax. This is a linguistic model, developed by Pieter Seuren as a direct continuation of Generative Semantics, a theory that flourished in the late sixties and early seventies. Seuren, however, chose to give his own model the more appropriate name Semantic Syntax, since it deals primarily with syntax, not semantics (though semantic considerations do play an important role).

As a syntactic theory, it aims at describing the grammatical arrangement of words within natural language sentences. Thus, it is concerned with questions like: which word orders are allowed? How should a sentence be divided into groups of words that behave as a unity? How do these groups of words (or constituents) relate to each other? Like most contemporary syntactic theories, Semantic Syntax represents the constituent structure of sentences by means of tree structures, which were introduced into linguistic theory by Bloomfield (1933). An example of such a tree structure is given in Figure 1.1. This tree represents a syntactic analysis of the sentence in (1). Note that, compared to a real-life tree, it is upside-down in that its root is at the top, with branches going down and leaves at the very bottom.

(1) Jim did not kiss Mary.

---

1 The tree in this figure represents an analysis in accordance with Semantic Syntax. Other syntactic theories may assign different tree structures to the same sentence – for example, a structure in which kiss and Mary form a constituent together. In Chapter 4, I will go into Seuren’s reasons for assigning this particular structure. Here, it merely serves to illustrate the general concept of tree structure.
The leaves are the actual words in their correct order (reading them from left to right results in the string in (1)). The branches and intermediate nodes represent the structural characteristics of the sentence and its constituting parts. Thus, this tree tells us that the whole string is a sentence (S), that did and kiss are verbs (V), not is an adverb (ADV), and Jim and Mary are noun phrases (NP). The features <SUBJ> and <OBJ> with the two NPs indicate that Jim is the subject of the sentence and Mary the object. Furthermore, the tree shows that the words did, not, kiss and Mary together form a constituent, called verb phrase (VP). That these four words form a syntactic unit is plausible because, for example, they can be co-ordinated with another VP, as in (2). Also, they can be replaced by another VP – for example, a VP with no object as in (3a), or a VP with two objects as in (3b).

(2) Jim [VP did not kiss Mary] and [VP will not hug Joan].
(3) a Jim [VP did not sleep].
    b Jim [VP did not give Mary a kiss].

Obviously, a tree structure is not something which is physically present in the human’s mind (or anywhere else), and which can be established objectively by putting it under a microscope. Thus, an important aim of a syntactic theory is to find criteria for determining what constitutes the best structural analysis of sentences from various points of view. From a descriptive point of view, it should tell us which analysis does justice to the native speaker’s intuitions about the acceptability of sentences. For example, why can one negate a sentence like (4a) by inserting not in front of kissed, as in (4b), while this is not possible in a sentence like (5a)? After all, the correct negation of (5a) is not (5b), but rather the sentence we saw in (1) with the auxiliary verb did.²

(4) a Jim has kissed Mary.
    b Jim has not kissed Mary.

² An asterisk in front of a sentence, as in (5b), indicates ungrammaticality.
(5)  a  Jim kissed Mary.
b  * Jim not kissed Mary.

From a crosslinguistic point of view, a syntactic theory should provide us with analyses that explain why languages do not only differ in their vocabulary, but also structurally. Taking the same example, why does English not allow simple insertion of the negation in sentences like (5a), whereas Dutch does allow this, as shown in (6a) and (6b)?

(6)  a  Jim kuste Marie.
     Jim kissed Mary
b  Jim kuste Marie niet.
     Jim kissed Mary   not

From a psycholinguistic point of view, a sound syntactic theory should provide analyses that are plausible if one takes into account various facts that are known from psychological study of human language processing. For example, how can it be that children very rapidly learn to use their mother tongue creatively in that they produce grammatical utterances they have never heard before? And why do these children make other mistakes than adult foreigners learning the same language?

Two decades after Bloomfield’s introduction of tree structures into linguistic theory, it was Harris (1951) who introduced the concept of a generative grammar. He argued that the preferred tree structures should be the ones defined by the simplest set of generative rules. For many syntacticians, it became the main aim to formulate a finite set of rules (a grammar) that would specify, by means of algorithmic generation, all and only the well-formed sentences of a particular language. Since the set of well-formed sentences of any natural language is infinite, a grammar must contain some form of recursion, allowing the rules to re-apply to their own output.

When determining what should be the preferred structural analysis, Semantic Syntax also takes semantic considerations into account. In this respect, it differs from related syntactic theories within a framework often referred to as Autonomous Syntax (see Seuren 1972). Semantics can be defined as the study of meaning – it is concerned with the principles that govern the relationship between sentences or words and their meaning.

A semantic analysis of a sentence can be represented in the formal language of first-order predicate calculus as developed within the modern study of logic. In this language, a proposition (or sentence) primarily contains a predicate with a number of arguments. Other parts of the meaning of a sentence, like tense and negation, are represented as operators with scope over this predicate. The predicate-argument structures with their operators can be represented as tree structures, too. Thus, a semantic representation of the sentence in (1) is given in Figure 1.2. This tree demonstrates that the sentence refers to a situation with an act of kissing (the predicate) with an agent (Jim) and a patient (Mary); the sentence states that somewhere in the past (the tense operator) this situation was not true (the negation operator).
The model of Semantic Syntax aims at relating syntactic structures as the one in Figure 1.1 to semantic representations as in Figure 1.2 by means of a transformational grammar. Transformational rules were introduced into the study of syntax in the late fifties, when linguists started to realize it would be effective to generate sentences in stages. In that way, one could derive, for example, both active and passive, declarative and interrogative sentences from one and the same basic structure. The general model became one with two modules: first, a context-free grammar generates so-called deep structures, which are then transformed into surface structures by a transformational grammar.

Semantic Syntax as described by Seuren (1996) is such a two-stage, transformational model. The deep structure is a semantic analysis (SA) of the sentence under consideration. This SA is essentially a predicate-argument structure as shown in Figure 1.2, albeit with other labels at the nodes. It is defined by a set of formation rules that draw on a lexicon. The SA is input for the main part of the Semantic Syntax model: the transformational rules. These can be divided in two. First, a set of cyclic rules operates bottom-up on each successive S-cycle, and second, a set of post-cyclic rules operates on the whole tree. The output of the transformational grammar is a surface structure (SS) as in Figure 1.1, with all the constituents in their right linear order, and enough information to be the input for a morphological and a phonological component. Schematically, the model looks as shown in Figure 1.3.

Seuren gives a detailed description of the model, formalized into generative grammars for a number of different languages. I will discuss these grammars extensively in Chapters 3 to 5 of this thesis. Now, I will turn to the questions regarding Semantic Syntax that we try to answer in this study, as already hinted at in section 1.1.
1.2.2 Research questions

There are two main questions regarding Semantic Syntax in its present state that we aim to answer in this thesis. They relate to the model’s descriptive adequacy and its crosslinguistic adequacy, respectively (see section 1.2.1).

On a descriptive level, Seuren (1996) points out a substantial number of grammatical constructions and phenomena that he claims to cover in the Semantic Syntax model. However, the question arises whether these claims can be substantiated to the full. The Semantic Syntax grammars are very intricate rule systems, in which the majority of the rules interlock like cogwheels in a complex machine. The individual rules are, for the most part, not too difficult to understand, but the way they interconnect is not always immediately evident. It is not easy to see at a glance which rules are involved in the generation of a particular sentence, let alone whether they generate the desired structural analysis.

In this thesis, we will test whether the grammars do indeed generate all the well-formed sentences that Seuren describes, and whether they exclude all the ill-formed cases as mentioned by him. We want to ascertain that the rule systems contain no inconsistencies. And if they do contain inconsistencies, we will examine if these can be solved within the model. We will also investigate whether none of the rules contain any unwanted or unexpected side effects.

A related practical problem concerns updating the Semantic Syntax grammars. This arises when one wants to extend the coverage of a particular grammar, or improve its adequacy or economy on the basis of new theoretical insights. Because of the intricacy of the grammars and the interaction of the various rules, it is very difficult to
foresee all the consequences of any change in an existing grammar. After such a change, one may of course manually go through all the transformational operations for a few relevant cases, but that is a time-consuming task and there is no guarantee that one has not overlooked some unexpected situation that did not occur in the test cases. This is a very real and practical problem that needs some solution if one seriously wants to develop the Semantic Syntax model any further, in order to extend its descriptive power.

On a crosslinguistic level, there is a second question we will seek to answer. Upon close study of Seuren’s grammars for the various languages, one will notice they have a lot in common. This is in line with the way these grammars were developed:

“The method followed bears all the bootstrapping characteristics of scientific induction. After thorough inspection of large sets of data in a few languages, tentative deceptions and analyses were developed. These became more and more daring as they proved successful. It was then felt that the grammars thus devised were cutting the patterns of the languages concerned ‘at the seams’. Thus it became time to try and make the formalism used more precise and more unified. As this book will make clear, we are, at the moment, in the middle of that process.”

(Seuren 1996, p.17)

Thus, in constructing a grammar for one language, Seuren kept adjusting and fine-tuning previously constructed grammars for other languages in order to maintain as much parallelism as possible. However, it is not always clear which of the similarities are structural, and which are merely a coincidence. In our view, the Semantic Syntax grammars call for theoretical generalizations with respect to the interrelation between them. Which aspects of the rule systems are language-specific, and which other aspects hold for groups of languages? If there are differences between the grammars of two languages, are they connected to each other as manifestations of one and the same deeper divergence? Referring to the quotation above, in trying to answer these questions we will continue the process in the middle of which Seuren considers the Semantic Syntax model to be.

In sum, this thesis focuses on the descriptive and crosslinguistic adequacy of Semantic Syntax. In the next section, I will explain how we intend to tackle the above-mentioned questions (section 1.3.1) and state more precisely what we will and will not involve in our search for answers (section 1.3.2).

1.3 Methodology

1.3.1 Technolinguistics

Our approach falls within the branch of linguistics known as computational linguistics. Computational linguistics is a term covering a broad range of research. It may be roughly divided into two distinct types of research, each with their own objectives and methods. The first type is what is often referred to as language technology. This
covers various kinds of research in which the main aim is to build systems that can deal with the complexities of natural language. Linguistic theory is one of the knowledge sources being used in this type of research, but all kinds of other resources (for example, statistic information, artificial intelligence techniques, insights from computer science) are applied as well.

The second branch of computational linguistics has a different focus. In this type of research, linguistic theory constitutes the central aim, and computational or technological means are used to further develop theories about natural language. This is what we call technolinguistics, a term introduced by Coppen (1991a). Technolinguistics is typically concerned with implementing linguistic models:

"Technolinguistics, as I see it, is concerned with the implementation of some formalization of some linguistic theory. This is not to say that a Technolinguist cannot make contributions to linguistic theory. What is meant here is that Technolinguistics commits itself to make implementations. This means that formalizations that cannot be implemented will be rejected, as will linguistic theories that cannot be formalized." (Coppen 1991a, p.4)

Thus, in the technolinguistic view, a linguistic model ideally consists of three separate components, as shown in Figure 1.4. At the basis, there is a linguistic theory. Then, if the theory is a sound one, it should be possible to derive a formalization from it. Finally, for any sound formalization one should be able to make an implementation.

Looking at Semantic Syntax from a technolinguistic point of view makes it clear that Seuren (1996) focuses mainly on the formalization part. The various grammars he describes in great detail for English, French, Dutch and German may be regarded as the formalization of underlying theoretical principles (see Figure 1.5). Some of these principles are explicitly formulated by Seuren, but the bulk of his work is devoted to the formal grammars. It is claimed that this formalization is precise and compact enough to be implemented straightforwardly:
"This information [on nodes, trees and operations on trees - L.T] will be provided in terms that may seem unnecessarily close to implementation. Their point, however, is not the imposition of some specific class of possible implementations but only, more modestly, to provide a set of notions precise enough to describe and analyse the operations occurring in the grammars of the languages described to an acceptable degree of formal precision." (Seuren 1996, p.45-46)

Given the questions put forward in the previous section, a technolinguistic approach appears to be a suitable and promising way to search for answers. In our view, an implementation is the obvious means to tackle the descriptive question. A computer program based on Seuren’s grammars is the ultimate way to test whether these grammars do indeed generate precisely the set of sentences that he claims they generate. Building a computer program will decisively reveal any inconsistencies within the rule systems, and it will force us to try and solve them. It will also bring to light whether the rules contain any unwanted side effects. Furthermore, if we indeed arrive at a sound technolinguistically motivated implementation, derived from the grammars as directly as possible, it will be a valuable tool for any linguist who wants to further develop the model of Semantic Syntax. For she or he can immediately inspect the results of adding or changing a rule within one of the grammars, by re-running the program afterwards. Since Seuren’s grammars are formulated as generative rule systems, our primary implementation will be a generator, i.e. a computer program that generates sentences.

The crosslinguistic question, too, will be approached in a technolinguistic way, namely by trying to separate formalization from theory. We will attempt to formulate the principles underlying the Semantic Syntax grammars more explicitly than Seuren does, and thus search for more generalizations. The crosslinguistic generalizations found will be reflected as parametrizations, both in the formalization and in the implementation. We will establish which differences between the grammars are purely language-specific, and which are a matter of parametric variation. These parameters will be integrated within the implemented generator. Additionally, we will also investigate how to build a parser, i.e. a computer program that analyses sentences. This, too, will help us in formulating theoretical generalizations. After all, since a parser cannot be derived directly from Seuren’s generative formalization, implementing one will force the researcher to look at the underlying theoretical principles.

In sum, looking at Figure 1.5 once more, we take Seuren’s Semantic Syntax grammars as our starting point and from there we proceed in two directions. On the one hand, we follow the lower arrow by implementing a generator. On the other hand, we re-trace the upper arrow by searching for theoretical principles, generalizations and parameters.

1.3.2 Delimitations

The main delimitations of the present study are prompted by Seuren (1996). As explained in section 1.2, the object of this research is Semantic Syntax. Given our
research questions and our technolinguistic methodology, we will aim to implement exactly what Seuren has formalized. Thus, our generator should precisely cover the set of sentences that Seuren implicitly or explicitly claims to describe. A representative selection of this set is given on the CD-ROM accompanying this book. It includes phenomena like tense, modals, datives, passives, questions, negation, adverbs, quantification, control structures, clitics, counterfactuals, and word order. In short, Seuren focuses on sentential and V-cluster syntax, while leaving things like NP-internal syntax and binding facts largely out of account. It is not our intention to extend the coverage of Semantic Syntax to linguistic phenomena that Seuren does not cover. In other words, whereas we aim to test the model’s descriptive adequacy we do not aim to expand its descriptive power.

Furthermore, we will limit ourselves with respect to the number of languages. Seuren (1996) presents detailed grammars for English, French, Dutch and German. Additionally, he also pays attention to Turkish and, to a lesser extent, to languages as diverse as Finnish, Malay and Italian. In a way, the grammars of each of the studied languages are complex enough to form the subject of a substantial investigation by themselves. In order to answer the crosslinguistic question, however, we obviously need to include more than one language in our study. Thus, we have decided to focus on English, Dutch and German. The main reason for selecting these three languages is that they promise to be an interesting set from the crosslinguistic point of view, given that they are historically related to each other. The grammars of the three languages as designed by Seuren show a great deal of resemblance to one another. Hence it will be likely that we can indeed implement a generalized grammar for the three languages together. On the other hand, the three grammars also show enough differences to be grounds for some interesting parameters.

In addition, a pragmatic reason for choosing these particular languages is our own familiarity with them. Since Dutch is our native tongue, we can fully rely on our own grammaticality judgements with respect to Dutch sentences. For judgements with respect to English and German, we have direct access to native (and near-native) informants in our research environment.

Thus, our primary implementation will be a combined generator precisely producing the set of English, Dutch and German sentences that Seuren (1996) claims to describe. A recapitulation of this set is given on the CD-ROM. Note that this set of sentences can also be used as a test suite by anyone who wants to further develop the Semantic Syntax grammars for these three languages. A change in the grammars can be evaluated by means of re-running them to generate the same test suite.

Our secondary implementation, the parser, will be far more limited. Studying Semantic Syntax parsing only serves to discover additional theoretical generalizations,

---

3 If the book has been delivered to you without accompanying CD-ROM, please contact the author at lteunissen@ntu.nl to receive a free copy of it.
so we have decided to restrict ourselves to just one language in this respect. For purely pragmatic reasons, the language chosen will be Dutch. This choice means that we can make use of extensive experience with Dutch parsing within our technolinguistic research group and of readily available Dutch parsing tools.

To conclude this section, let me say something about what we will not do in this thesis. First of all, we will not compare the model of Semantic Syntax to other syntactic theories. Although comparison to, for example, other generative models like the well-known Autonomous Syntax theories from the Chomskyan school (Government & Binding, Principles & Parameters, Minimalism; see, for example, Chomsky 1982, Chomsky & Lasnik 1993 and Chomsky 1995), or to unification models like Head-driven Phrase Structure Grammar (see Pollard & Sag 1994) might be a meaningful and fruitful way of evaluating Semantic Syntax, this would be a complete study in itself. From a comparative point of view, Seuren comments on the relation between Semantic Syntax and Chomskyan theories as follows:

"Needless to say, Semantic Syntax differs fundamentally from the developments in Autonomous Syntax in every conceivable way. It has, at least in its present state, no X-bar theory and no traces and, in general, none of the constraints now current in MIT-based syntactic theory."

(Seuren 1996, p.14)

Although there are indeed fundamental differences between the two theories, mainly with respect to the methodology and terminology, it is also striking that at a certain level some of the actual analyses are in fact quite similar (in our view, several of the recent developments in the Minimalist Program are highly reminiscent of Semantic Syntax operations). But we will leave it to the reader to draw her or his own comparative conclusions in this respect, because we agree with Seuren that:

"if Semantic Syntax is to be set off against the latest developments at MIT, other theories of grammar are equally entitled to a careful and detailed discussion, and this would require a volume in its own right."

(Seuren 1996, p.16)

Second, what we will also not engage in are psycholinguistic considerations. In choosing the right implementation of a particular rule, we will only be guided by syntactic and semantic criteria, not by psychological models. And third, we will also restrict ourselves to the modern standard varieties of the three languages mentioned. Thus, we will not attempt to account for certain dialectal variations, nor for older stages of the language.

---

4 In a review of Seuren (1996), Denis Bouchard takes a similar stand: "Despite the superficial differences due to the fact that this framework appears to be frozen in time, as is reflected in its terminology and its theoretical tools, the results it obtains and the way in which they are obtained are eerily similar to what is found in current approaches like Principles & Parameters and the Minimalist Program. [...] The advocates of Semantic Syntax and of Principles & Parameters emphasize that their conceptualizations of grammatical theory are highly different, the former praising the 'old values', the latter dwelling at length on the conceptual gains made in the last thirty years (this is crystallized in the introductions of Semantic Syntax and of Chomsky 1995). But why is it that, if we concentrate on the mechanisms that actually make the analyses work rather than the surface presentation of the theories, the end results are so similar?" (Bouchard 1999, p.219).
1.4 Related research

1.4.1 Semantic Syntax

Seuren

As the reader will have understood by now, the main work on Semantic Syntax is the book published by Pieter Seuren in 1996. Besides a detailed discussion of the backgrounds and the formal properties of the Semantic Syntax model, the greater part of this book is taken up by four chapters describing the grammars of English, French, Dutch and German, respectively. The intricacies of these grammars will be discussed throughout the remainder of the present thesis, so I will not go into any details here.

The 1996 book should not be confused with an earlier volume, edited by Seuren and also titled Semantic Syntax (Seuren 1974). This volume is a collection of papers from various linguists within the framework of Semantic Syntax alias Generative Semantics. For, as stated before, the modern model of Semantic Syntax can be seen as a continuation of the tradition of Generative Semantics, a theory that had its heyday in the late sixties and early seventies. I will not discuss this tradition here, however, since its literature from those days does not contain anything like a worked-out rule system as yet. And the rule system is what we are primarily interested in. The reader who wants to know about the earlier issues and discussions within Semantic Syntax and Generative Semantics is referred to the articles collected in Seuren (1974) and McCawley (1973), for example.

Seuren kept working on Semantic Syntax throughout the seventies and eighties (see, for example, Seuren 1976, Seuren 1984, Seuren 1986), but it was not until the nineties that the Semantic Syntax rule systems gradually took their present shape. Besides the 1996 book, Seuren wrote a few articles on specific aspects of the model in this period. In 1993, he presented a paper at the yearly Computational Linguistics in the Netherlands meeting (Seuren 1994). This paper focuses on translation relations between Semantic Analyses (see Figure 1.2, p.5) in different languages. Other papers that we will refer to in this thesis are unpublished manuscripts on verbal clusters in German and Dutch (Seuren 1997) and on adverbs in English (Seuren 1999b), and an article on branching directionality (Seuren 1999a).

Although Seuren is now a lone representative of the Generative Semantics a.k.a. Semantic Syntax framework, the 1996 book has generally been received rather well. A number of reviewers have recognized it as a valuable contribution to generative grammar, despite its somewhat unusual terminology and base assumptions, and despite the apparent lack of awareness of current analyses (especially those within the Chomskyan framework, but see also our short discussion of this in section 1.3.2):
“This is a very well presented and remarkably detailed description of English, French, Dutch, German, and Turkish at the sentence level. It exemplifies a long-standing theoretical claim which has gained ground nowadays, namely that the structural descriptions of linguistic objects should stand for both semantic and syntactic information. This contrasts with the widely adopted practice of analyzing sentences in syntactic terms only.” (Markantonatou 1998, p.664)

“This book covers a vast array of constructions which are at the forefront of current theoretical discussions. [...] Semantic Syntax may be a very useful tool to measure the progress made in linguistic theory, and to identify areas where things got stuck.” (Bouchard 1999, p.222)

“Semantic Syntax is, as implied above, a remarkably impressive statement of the present position of a major, though currently unfashionable, form of TGG. Seuren presents SESYN as just a step along a road that will hopefully lead to a system of universal principles, a goal shared by all TGG theorists. He is nevertheless clearly satisfied with its performance in accounting for the languages to which it has been applied so far. For these reasons, Semantic Syntax constitutes an important contribution to grammatical formalism and syntactic theory.” (Longa 1999, p.93)

Schotel

The crystallization of the Semantic Syntax grammars in the nineties was substantially aided by work done by Henk Schotel, who developed a workbench for Semantic Syntax (Schotel 1994). This workbench, called SeSynPro, is a Prolog program that was meant as a tool for testing and developing the model and for teaching it to students. Seuren’s grammars and Schotel’s program were developed on a par, in close collaboration, following an approach that Schotel calls the computational modelling research scheme:

“Although the basic scheme of the theory is very simple, the details required to make Semantic Syntax descriptions fit the complexities of a natural language made it worthwhile to find ways to help the research process, which until very recently was done without the use of computers [...]. The implementational work resulted in the discovery of inconsistencies and lack of detail (by HS), which were fed back (to PS) to improve the theory. This is an ongoing process, no details of this were recorded, but particularly in the post cycle there were several occasions where the computational modelling research scheme payed off. The workbench will facilitate the application of the descriptive methods of Semantic Syntax to other languages than the ones from which it was induced [...]. Such descriptive efforts will undoubtedly lead to the discovery of other shortcomings and give rise to further improvements of the theory. So the workbench can be considered as a tool resulting from a linguistic theory, leading to a faster development of that theory.” (Schotel 1994, p.136)

Obviously, there are parallels between Schotel’s work and the present study. In technolinguistic terms, Schotel too has made an implementation based on Seuren’s formalization. His aims as cited above are reminiscent of our first research question as to the descriptive adequacy of Semantic Syntax.

However, in our view, there are a number of significant differences as well. First of all, Schotel has not formulated any explicit questions to answer. And, probably related to this, his approach (the computational modelling research scheme) seems to be not as systematic and methodologically well-founded as our technolinguistic approach. More importantly, as Schotel admits in the citation above, the results of his implementing efforts were not recorded systematically. Thus, the reader cannot judge for her/himself in which respects the implementation indeed has contributed to
the formalization. Furthermore, our second research question as to the model’s crosslinguistic adequacy has not been addressed by Schotel at all.

Perhaps the most important difference is the implementation itself. Schotel has used Prolog, a general-purpose programming language, to implement the Semantic Syntax grammars. As a result, linguistic issues and aspects of the applied algorithms are mixed with each other. Because of this syntax-embedded approach, it is not always clear which parts of the program correspond to which grammar rules. In contrast, in the light of our specific research questions and technolinguistic methodology, we will use special-purpose tools to implement the Semantic Syntax model and always aim at a one-to-one relationship between Seuren’s formalized rules and our implemented rules. Additionally, Schotel (1994) has paid much attention to a graphical user interface that facilitates the drawing and editing of trees. Because of this feature, the workbench is very well suited for linguists and students who want to familiarize themselves with the Semantic Syntax model – it allows them to experiment with deriving particular sentences according to the implemented grammars. Our own implementation, on the other hand, will be better suited for linguists who want to develop the Semantic Syntax model themselves – they can experiment with the implemented grammars by adding, changing or removing rules. We will only supply a rudimentary user interface. See Chapter 2 for more details about the particular choices we will make with respect to our (primary) implementation compared to other systems.

In sum, whereas “SeSynPro is the first attempt to implement Semantic Syntax” (Schotel 1994, p.135), the present work can be regarded as a second, more systematic and well-documented attempt to do so, guided by explicit linguistic research questions and following technolinguistic methodology, without paying specific attention to the user interface.

**Richter**

Last year, Michael Richter finished his PhD thesis on the German verbal system (Richter 2000). Besides a thorough study of the extensive literature on this subject, the thesis presents a Semantic Syntax analysis of a number of phenomena regarding the verbal system in German. As such, it can be seen as a detailed supplement to Seuren (1996) on this specific subject. And thus, we will also include Richter’s findings in our implementation wherever applicable (see, for example, section 3.2.3).

In fact, Richter has been the first linguist to use our implementation for the purpose of further developing the Semantic Syntax model. In his chapter 3, he explains how he has used and adapted our system to generate the German verb constructions that he accounts for:

“As an illustration, I will now show how a number of the syntactic principles as discussed in this work can be turned into a generating computer program. [...] I will discuss the
implementation of transformational rules which account for essential syntactic constructions as treated in the present study.”  

(Richter 2000, p.179-180 – my translation)

Although Richter’s discussion of the implementation is relatively short and therefore somewhat sketchy and simplified, it becomes clear that his findings can indeed be integrated into our generative implementation. In accordance with the technolinguistic view that any sound formalization should be implementable, he takes this as an indication that his contributions to the Semantic Syntax model are consistent with it:

“Furthermore, the successful implementation of the rules into the system, which is designed in a strictly parallel way to Semantic Syntax, shows that the theoretical model as well as its computational translation are coherent.”  

(ibid, p.198 – my translation)

Thus, Richter’s efforts show that our implementation can indeed serve as a tool for the development of Semantic Syntax. I should add, however, that at the time that Richter worked with our system, it was not yet finished. We have made several important alterations to it since, independently of his work. Therefore, it will not be possible for us to simply copy Richter’s suggestions for the implementation of his analyses. We will have to ascertain whether they still fit in with the current system.

1.4.2  Technolinguistics

Coppen

As explained in section 1.3.1, the technolinguistic terminology and methodology are due to Coppen (1991a). In his thesis, Peter-Arno Coppen links the (autonomous) syntactic theory of Government & Binding to the semantic theory of Montague Grammar within the area of the Dutch noun phrase.

“The aim of technolinguistics is to develop a kind of formal model that can be implemented straightforwardly, while, at the same time, it is clearly based on current linguistic theory. An implementation of such a formal model should be able to relate structural descriptions [...] to natural language sentences [...]. As such, implementations may be developed that generate utterances (i.e. they connect structural descriptions with sentences), or that analyze sentences (i.e. they connect sentences with structural descriptions). In my view, it is not an a priori that these two implementations should be similar. The only prerequisite is that they are both based on linguistic principles. That is, both generation and parsing should be principle based. Also, this does not mean that the formalization needed to implement linguistic theories should be in any way similar to the formalization already undertaken by GB theory.”  

(Coppen 1991a, p.5)

---

5 The methodology has its roots within work done on the AMAZON/CASUS system as initiated by Jan van Bakel in the late seventies and early eighties (see van Bakel 1983, van Bakel 1984). That the present study falls within the same tradition as this work becomes clear from the following statement by van Bakel: “When testing a semantic-syntactic theory of natural language sentences, the computer simulated theory should be required to assign the correct syntactic structure to the sentence and also to express the corresponding meaning.” (van Bakel 1983, p.... – my translation). We will use the AMAZON module in our own parser (see Chapter 6).
Thus, as a technolinguist, Coppen first separates theory from formalization within both frameworks and then makes his own formalizations on the criterion of implementability. Additionally, he describes how he has implemented these formalizations. He argues that, besides being principle-based, a technolinguistic implementation should also be modular:

“In this [technolinguistic] model, grammatical analysis is performed by means of independent modules, connect to each other. The output of the first module serves as input to the second module.”

( Ibid, p.293)

We will follow Coppen in this view on technolinguistic methodology as well. As will become clear in Chapter 3 and Chapter 4, both Seuren’s formalization and our implementation of Semantic Syntax are divided into a number of separate modules. Other publications by Coppen on the technolinguistic approach are Coppen (1992), Coppen & van der Ende (1993) and Coppen & van Bakel (1994).

B. van Bakel

Bas van Bakel (not to be confused with his father Jan van Bakel, as cited in footnote 5 above) also follows an explicit technolinguistic approach within his PhD thesis (van Bakel 1996). He focuses on the principle of linguistic motivation as the central notion within technolinguistics:

“This principle can be defined as follows: [...] In Natural Language Processing, solutions to computational problems must be motivated by general linguistic considerations. The LM Principle holds at all levels within NLP. It is based on the idea that only linguistics can provide a sound and substantial base for natural language processing, and will lead to maintainable and expandable systems that perform well.”

( van Bakel 1996, p.3)

The principle of linguistic motivation also holds for the present study. After all, in an implementation that is meant to be a tool for testing and developing a linguistic model, it is only logical to turn to this model for the solution of problems encountered.

The main difference between van Bakel’s study and the present work is the objective. Whereas our aims are purely linguistic in that we try to answer linguistic questions, van Bakel shows that the technolinguistic approach also pays off in application-oriented research. His objective is to create a system for thematic analysis of English, which will become part of an information extraction system for chemistry texts. Taking an instrumentalist view, he demonstrates that it makes sense to use linguistic theories in application-oriented studies.

1.5 Conclusion

We can conclude that in the present study two traditions come together. As to our research object, this work belongs to the tradition of transformational generative
grammar, more specifically that of Semantic Syntax. We have committed ourselves to answer two types of questions with respect to the Semantic Syntax model. First, we want to know whether the Semantic Syntax grammars are descriptively adequate in that they generate exactly the set of sentences that they are claimed to generate; a connected practical problem that we want to solve is that the grammars have become too complex to ensure descriptive adequacy when one adapts them in any way. Second, we want to try and take the Semantic Syntax model to a higher level of crosslinguistic adequacy in that we will search for theoretical generalizations about the interrelation between the grammars for various languages.

As to our methodology, this work belongs to the tradition of computational linguistics, more specifically that of technolinguistics. This implies that we will build a principle-based, modular implementation from Seuren's formalization on the one hand and that we will try to separate the formalization from the underlying theory on the other.

The main part of this thesis describes our primary implementation, the generator. First, I will discuss the field of natural language generation from a number of perspectives that are particularly relevant to our technolinguistic approach (Chapter 2). Then, I will go into the two smaller modules of our generator, the formation rules and the lexicon as implemented for English, Dutch and German (Chapter 3). In the next two chapters, I will describe the main module: the transformational grammar. Chapter 4 focuses on the implementation of the transformational rules for English. In Chapter 5, I will explain how we have adapted this implementation for the other two languages, using parameters.

Finally, Chapter 6 describes our secondary implementation, the parser. I will describe the general make-up of a basic Semantic Syntax parser for Dutch as we envisage it. This chapter is meant to shed some additional light on the descriptive question, mainly from the technolinguistic goal of bringing to light the theoretical principles behind a formalization.
CHAPTER 2  

Natural language generation

2.1 Introduction

The field of natural language generation deals with the task of automatically changing some kind of information representation into an adequate text. The information representation is, for example, (part of) a database or the output of another computer system. The created text may consist of one sentence or of several paragraphs; it is either issued in written form on a computer screen, or sent to a speech synthesizer.

Research in the area of natural language generation began around 1970, as a spin-off from natural language processing. Bateman and Hovy (1992) give a fairly minute survey of the problems and principles of language generation, outlining some of the main systems developed in the seventies and eighties. I will not try to do the same here; instead I want to discuss the field from a few angles that are particularly valuable for our own research, using, whenever possible, more recent systems as examples.\(^5\)

Language generation research is usually a blend of linguistics, artificial intelligence and psycholinguistics. The interdisciplinary nature is reflected in a number of attributes on which the various projects differ:

- the purpose for which the generator is built (theoretical or practical);
- the task of the generator (grammatical realization or text planning);
- the method of generation (ranging from canned text to feature systems);
- the tools for implementing the generator (general programming languages or tailor-made environments).

---

\(^5\) In choosing examples, I will give preference to Dutch systems, as they generally do not get as much attention in the literature as the English language generators. However, this chapter is of course not meant to give a complete survey of the state of the art of natural language generation in the Netherlands.
2.2 Aspects of natural language generation

2.2.1 Purpose

A natural language generator may be built with either theoretical or practical intentions, or something in between.

Many systems are built for purely theoretical ends. In linguistic studies, generators are built to test various hypotheses with respect to syntax, semantics, pragmatics and other areas. An example is the system developed by Marsi (1995) to study prosody in Dutch. Marsi, who can be considered as belonging to the technolinguistic tradition as well, stresses that his system is meant to be a research tool; as such, it “provides a way to verify hypotheses about the relation between discourse, semantic and syntactic structure on one hand and prosodic structure on the other” (Marsi 1995, p.85).

In psycholinguistic studies, making a generator can be quite useful to model the human language production process. This is done, for example, by De Smedt (1990) who has implemented an incremental generation model, based on earlier work by Kempen and De Smedt in the early eighties.

“By means of computer programs, psycholinguistic processes can be simulated with a precision and explicitness as never before. Even if it is not possible to prove, by constructing a computer program, that a theory is adequate or valid, it may be possible to discover flaws in the theory by analyzing the program and comparing its behavior with experimental data.”

(De Smedt 1990, p.7)

Other systems are built with only practical purposes in mind. Generators are, for example, needed in dialogue systems, where the computer has to formulate a natural language response to a user query (or a natural language question needed to get information from the user). A Dutch example of this is the OVIS2 system, an automated telephone inquiry system for public transport information (developed within the ARIS project, see Strik et al. 1997).

Another important area in which natural language generation is applied is that of machine translation: systems that deduce some kind of meaning representation from a sentence in the source language also need a module to convert this representation into a sentence in the goal language.

The motivation for building a generator may have important consequences for the design of the system. For example, the lexicon or dictionary that the system uses will depend largely on its purpose. Systems with practical goals usually need to deal with large lexicons, whereas those with theoretical goals are often equipped with relatively small toy lexicons.
2.2.2 Task

Most generators focus on one of two tasks:

- text planning (the question of what to say, also referred to as strategic generation or content selection);
- grammatical realization (the question of how to say things, also referred to as tactic generation).

Traditionally, most language generation research has focused on the grammatical realization task. The majority of the systems have been (and perhaps still are) sentence generators: they produce single sentences, and are usually not concerned with larger textual issues, nor with content selection. Insights for this type of generation research have been supplied primarily by grammarians. An example of a generation system focusing on grammatical realization is the one developed by Schotel (1994), as discussed in section 1.4.1. His system produces single sentences and leaves lexical selection and other semantic choices to the user.

In the early eighties, some research effort began to be devoted to the job of generating more than one sentence. It immediately became clear that to create coherent text some kind of planning was needed. In this type of research the focus shifted entirely from grammatical realization to text planning; many of the insights were provided by philosophy of language. An example of a text generator can be found with Klabbers (1997), who describes a generator called GoalGetter. This system turns statistics on soccer matches into short commentaries. In the generation process, text planning is done to produce a coherent story. The story opens with some general facts about the match, then the information about the course of the match is conveyed in chronological order, and the story ends again with some general features. Another aspect of planning deals with delivering the right referring expressions. For example, if one player has scored two goals, he is first referred to by his name but later with an expression like “the centre forward”; the result is a more natural text.

Just as sentence generators ignore the task of text planning, most text generators do not pay much attention to the task of grammatical realization. GoalGetter, for example, uses the simple method of templates and canned text (see next subsection) to do this job, and some systems do not even try to make actual sentences but simply produce sentence specifications. Since most of the generation research is focused on one of the two tasks, problems that have to do with connecting the two are generally left unsolved. According to Bateman and Hovy (1992) this “generation gap” has only recently begun to be addressed.

2.2.3 Method

Natural language generators also differ with regard to the method they use in the generation process. Focusing now on the task of grammatical realization, and fol-
lowing Bateman and Hovy (1992), we can roughly divide the possible methods into four types (ordered from the most ad hoc to the most general):

- **canned text**;
- **templates**;
- **cascaded items**;
- **features or transformations**.

Many systems use **canned text**, which is quite efficient if the number of sentences to be produced is limited. However, these systems have no way of generalizing over sentences, and are very rigid. The canned text method is only useful if every desired output can be predicted. An example of canned text is provided by the messages that one comes across on the screen of a cash machine:

```
Welcome. Please insert your card.
```

When generated sentences consist partly of fixed material and partly of variable information, we speak of **templates**. A template is a canned message with one or more open slots that are filled by variable information. This method is very popular, and is used, for example, both in the Ôvis2 system (Strik et al. 1997) and in Goal Getter (Klabbers 1997).

If a system needs to be even more flexible, it can generalize over the templates and make use of **cascaded items**. Generation then becomes a process of repeated expansions. This approach can result in a phrase structure grammar. An example of this is the first module of the system developed by Schotel (1994). A related method is that of incremental sentence generation described by De Smedt (1990).

Generalizing even one step further, we arrive at a **feature** system. Features offer a method to abstract away from things like the order of constituents and to generalize, for example, over agreement patterns between subject and verb. An example of this can be found with Marsi (1998), who uses the Functional Unification Grammar feature notation framework, as developed by Kay (1979). The powerful process of unification performs the combination of features from different sources in such a way as to respect their joint requirements.

A related method that can be used is a **transformational** system. If the input for the generator is a tree structure, it can be transformed into another tree representing the surface structure of the sentence. A transformational system can be as general and as powerful as a feature-based system. An example of this is the second module of the system developed by Schotel (1994).

### 2.2.4 Tools

The tools that are used to implement a generator, can be roughly divided into two classes:
• general, all-purpose programming languages;
• tailor-made programming environments.

Many generators are implemented directly in a general programming language, like C or Prolog. These systems are usually syntax-embedded. Specific programming languages are chosen for their efficiency, their portability or other reasons. An example can be found with Schotel (1994), who has programmed his system in Prolog.

Other generators are implemented with tools that are expressly designed for such a purpose, for example a special grammar interpreter that acts as an interface between the developer and the program code. Such environments may be less efficient at times from a technological point of view, but they are usually better suited for building a generator that is insightful in that the underlying theory is separated from the implementation (syntax-directed). An example of this can again be found with Marsi (1998), who has used a special interpreter for the Functional Unification Grammar formalism. Another specific implementation tool that is used for natural language generators are the Augmented Transition Networks (see Bateman & Hovy 1992, p.58).

2.3 Technolinguistic approach to generation

2.3.1 Purpose

Our research questions (see section 1.2.2) and our technolinguistic approach to answer them (see section 1.3.1) have direct consequences for each of the four aspects of natural language generation discussed in the previous section. First of all, our purpose is primarily a theoretical one: we want to test and expand the theory of Semantic Syntax. In accordance with technolinguistic methodology, building a computer model of a linguistic theory is a test for such a theory. Since the theory of Semantic Syntax is formalized by Seuren (1996) as a set of generating grammars, our primary implementation of this will be a generator.

Having a theoretical purpose does not mean that the system cannot serve any other use. First, it will be a useful tool for anyone who wants to work within the Semantic Syntax framework. It will be of help not only to those who want to familiarize themselves with the grammars as described by Seuren, but also to those who want to expand these grammars by adding, changing or removing any of the rules. This is in fact the way Richter (2000) has already been using the system (see section 1.4.1). Since technolinguistic methodology dictates that every module and rule in our generator reflect a module or rule within the formalized theory (see also Coppen & van der Ende 1993), it will be relatively easy to have the generator keep up with any future adaptations to the grammars.
And second, the system may also be re-usable as a module in larger systems. This will hold especially for the transformational part of the system, which transforms semantic analyses into surface structures (see Chapter 4 and Chapter 5) and as such should be re-usable as a syntactic generator in language generation systems.

2.3.2 Task

The task that our generator has to carry out is defined by the model of Semantic Syntax and our questions with respect to it. Semantic Syntax is a syntactic model, which focuses on the grammatical relation between semantic structure and surface structure. Thus, the task of our generator will be grammatical realization. It is to produce sentences and model syntactic phenomena.

We will not pay attention to the problems of text planning, or to other aspects outside the domain of the syntactic generation of individual sentences. The generator will not be concerned with lexical selection or morphology, for example. The lexical selection will be done either randomly, thus leading to nonsense (though grammatical) sentences, or by the user of the system. The latter method is also known as directed generation (see section 3.1). The generator’s output is assumed to be fit for feeding it into a morphological component, but we will not concern ourselves with such a component in the present study.

2.3.3 Method

Because of our technolinguistic approach, our method is also directly derived from the model of Semantic Syntax. It is in fact a combination of two methods.

In the first module (see Chapter 3), we will use the method of cascaded items or expansion to create semantic analyses. This module will be a rewrite grammar that closely follows the formation rules as given by Seuren (1996).

The second module of the generator (see Chapter 4 and Chapter 5) will apply the transformational method to convert the semantic analyses into surface structures. This module, too, will closely resemble Seuren’s rule systems.

2.3.4 Tools

Technolinguistic methodology and the Semantic Syntax model also determine what kind of tools to use: tailor-made programming tools that have been developed specifically for implementing rewrite grammars and transformational grammars. The use of such tools ensures that we can make a syntax-directed implementation in which every module or rule is motivated from a module or rule within the formalized theory.
For the first module of the generator we will use the GRAMGEN system, which is an interpreter for Extended Affix Grammars (EAGs) developed by Coppen. Seuren (1996) has formalized his theory of a general SA-format by means of a set of formation rules, which are essentially context-free rewrite rules. He has devised his own notation for these rewrite rules. Our technolinguistic method implies that we should follow Seuren’s formalization, but not necessarily the notation used. Instead, with EAG we opt for a more generally used and computationally more elegant formalism. A pragmatic ground for our choice of GRAMGEN is the mere availability of it and of sufficient EAG expertise at the University of Nijmegen.

For the second module we will choose GRAMTSY, an interpreter for transformational grammars that has also been developed by Coppen (1991b). As explained in section 1.2.1, the formalization of the second component of Semantic Syntax has a transformational nature. From a technolinguistic point of view, implementation should therefore take place in the form of a transformational grammar. GRAMTSY reads and executes transformational grammars in a formalism strongly reminiscent of the notation widely used in the sixties. Several schools in generative grammar have tried over the years to impose all kinds of restrictions on the formalism, but none of these have had any essential consequences for its generative power. Chomsky’s (1995) Minimalist Program is just as transformational as McCawley’s (1973) Generative Semantics. A transformational system like GRAMTSY is thus in principle suitable (and, from a technolinguistic point of view, necessary) for implementing all these grammars, including Semantic Syntax.

Both GRAMGEN and GRAMTSY were developed at the Department of Language and Speech, at the University of Nijmegen. More information on these tools is given in Chapter 3 and Chapter 4, respectively, and also in the appendix. It would be an interesting future project to examine the possibility to re-implement the generator with the newer systems AGFL (developed at the Department of Computer Science at the University of Nijmegen) and TREMA (developed at the University of Twente). The latter tool is the one we will use for our basic parser (see Chapter 6).

2.4 Conclusion

To sum up: our purpose is theoretical, the task is sentence generation, the method is a combination of expansion and transformation, and the tools are custom programming environments. In the remainder of this thesis, I will refer to the implemented generator as GENIUSS, an acronym of Generative Implementation of Universal Se-

---

7 However, to express all relevant generalizations in the implementation, we will accommodate GRAMGEN with some extra mechanisms that do not belong to standard EAG (see the appendix).
mantic Syntax.\textsuperscript{8} The system is divided into three modules: the formation rules, the transformational grammar, and the lexicon. I will describe these parts in the following chapters.

Chapter 3 discusses the two smallest modules. After an introduction, it first discusses the formation rules as they will be implemented for the three languages under consideration, i.e. for English, Dutch and German. The rules will be implemented separately for each of these languages, but since they are highly similar I will describe them on a par. A separate section addresses the way we will implement the lexicon, which is used by both of the other modules. The next two chapters describe the main module: the transformational grammar. In Chapter 4, I will focus on the implementation for English. In Chapter 5, I will explain how we will adapt this implementation for the other two languages, using parameters. In all three of these chapters, the implemented rules will be introduced bit by bit. The reader can easily recognize them as they are presented in grey blocks, with new elements always in bold typeface.

Figure 2.1 is a schematic representation of the complete generator, in which the three modules can be identified from left to right as the lexicon, the formation rules, and the transformational grammar. Every box represents one of the files that together make up the generator. Thanks to our technolinguistically motivated strict separation of linguistic matters from the algorithmics, we can restrict the discussion to the few shaded boxes at the top of the figure, which depict the lexicon and rule files themselves. I will refer to these as the linguistic files. All of the other boxes are of no concern here, since they merely represent either the program tools that perform some action on the linguistic files (the boxes with rounded edges), or the intermediary files which are compiled from the linguistic files by these tools (the white boxes). The linguistic files are also laid out in full in the CD-ROM accompanying this book.

\textsuperscript{8} Obviously, the term universal should not be taken literally. I merely use this term to express that the generator will contain a grammar that generalizes over three languages, instead of three separate language-specific grammars.
Figure 2.1 *The three components of GENIUS*. 
CHAPTER 3  

*Formation rules and lexicon*

3.1 Introduction

The formation rules are implemented in the form of an EAG (Extended Affix Grammar, see the appendix). The EAG formalism makes it possible for us to extend the rules with a number of features and mechanisms that Seuren (1996) only describes in the text without formalizing them. Also it will turn out to be possible to express some extra generalizations in the EAG. Information that Seuren expresses by means of (sometimes slightly cryptic) super- and subscripts in his notation is projected by us onto affixes in the EAG. In doing so, we will try and find meaningful and theory-independent affix names whenever possible.

An EAG consists of two layers, with two context-free grammars laid on top of each other, so to speak. The bottom layer is the actual grammar and the top layer consists of the affix level. The interaction of both layers gives the formalism context-sensitive power. At first glance, a context-sensitive formalism may seem too powerful for implementing a context-free component, but in fact the two-layered structure is perfectly suitable to express both the general SA-format and its particular instances and conditions. The bottom layer expresses the general format, whereas the affix level contains all further information and restrictions. The domain of the affixes is defined in the metagrammar. By keeping the metagrammar finite, we will leave the context-sensitive power unused, so our EAG is in fact equivalent to a context-free grammar.

As can be seen in Figure 2.1, the relevant part of which is repeated below as Figure 3.1, the EAG is interpreted by GRAMGEN. This program tool delivers an SA, based on the formation rules, the lexicon, and user input. The user input ranges from just the number of SAs to be generated (fully random generation), to a complete specification of which terminals and non-terminals to choose at every point in the generation process (fully directed generation).

In the next section, I will describe the implementation of the formation rules. As already said in section 2.4, this will be an integrated description for the three languages under consideration. In section 3.3, I will describe the lexicon. Finally, in section 3.4, I will conclude what we have learned about the descriptive and cross-linguistic adequacy of Semantic Syntax by implementing its formation rules and lexicon.
3.2 Formation rules

3.2.1 The tenses

Central in Seuren's (1996) formation rules of English, Dutch and German are the following three rules:

\[
\begin{align*}
S'' & \rightarrow V_{t1} + S' \\
S' & \rightarrow V_{t2} + S^0 \\
S^0 & \rightarrow V_{LEX} + <lex.arg.frame>
\end{align*}
\]

The first two rules introduce the two tense operators, which are present at least once in all (standard) SAs. \(S''\) is a finite clause, in which \(V_{t1}\) determines whether the verb has present or past tense. \(S'\) has simple or perfect tense, as determined by \(V_{t2}\). The third rule introduces the lexical predicate, which is also an obligatory element in every SA. The general tense scheme is illustrated in Figure 3.2.
In the implementation, we use three affixes for this central piece of information: FIN, PERF and LEX, respectively. These affixes appear on both S and V. As opposed to the super- and subscripts used by Seuren, which do not express a direct relationship between S and V, the affixes in the implementation are identical for mother and daughter. This indicates that V is the head of its mother S. For all three languages, the relevant part of the EAG rules thus could be:\(^9\)

\[
\begin{align*}
S < \text{FIN} > & : \quad V < \text{FIN} >, \quad S < \text{PERF} > . \vspace{1em} \\
S < \text{PERF} > & : \quad V < \text{PERF} >, \quad S < \text{LEX} > . \\
S < \text{LEX} > & : \quad V < \text{LEX} > , \ldots
\end{align*}
\]

However, these rules obviously fail to express the generalization mentioned above, namely the fact that the affix of a head V is always identical to the affix of its mother S. To capture such generalizations, the EAG formalism supports the use of affix non-terminals (see the appendix).\(^10\) A unification mechanism sees to it that multiple occurrences of a non-terminal within one rule always receive the same instantiation. Using an affix non-terminal, we could combine the three rules into one general rule, plus a metarule to define the domain of the non-terminal:\(^11\)

\[
\begin{align*}
tense & : \quad \text{FIN; PERF; LEX.} \vspace{1em} \\
S < \text{FIN} > & . \vspace{1em} \\
S < \text{tense} > & : \quad V < \text{tense} > , \ldots
\end{align*}
\]

---

\(^9\) I will address the rewrite mechanisms for the lexical argument frame in the next section; for now, I leave this part unspecified in the rules.

\(^10\) Affix non-terminals may also be called variables or names, and affix terminals are sometimes referred to as values. Here, however, I follow the more formal nomenclature of Meijer (1986). When the context leaves no room for confusion, I will simply use the term affixes when referring to either non-terminals or terminals.

\(^11\) We keep to the convention that affix non-terminals are in lower case, and affix terminals in upper case. In Gramgen, the terminals need to be between double quotes, but here I will leave those out for the sake of readability.
Clearly, this implementation faces a problem: how do we encode what the sister nodes of the different instantiations of \texttt{V\langle tense\rangle} should be? In other words, how can we express the dependency between mother S and daughter V on the one hand, and daughter S (or lexical argument frame) on the other? To solve this problem, we have added a hierarchy notation to \textsc{Gramgen} that is not standard EAG (see the appendix). By means of angle brackets a metarule not only defines the domain of an affix non-terminal, but also the order in which the terminals are to be substituted for it:

\begin{verbatim}
type:: FIN << PERF << LEX.
tense:: FIN; PERF.
S\langle type\rangle.
S\langle tense\rangle: \texttt{V\langle tense\rangle}, S\langle type\rangle.
S\langle LEX\rangle: \texttt{V\langle LEX\rangle}, ...
\end{verbatim}

Now, the metarule for \texttt{type} expresses the obligatory sequence of S-expansions (and thus also the obligatory sequence of Vs). A separate rule is still needed for the expansion of \texttt{S\langle LEX\rangle}, since the sisters of \texttt{V\langle LEX\rangle} do not belong to the hierarchy but to the lexical argument frame (for the same reason the affix non-terminal \texttt{tense}, unlike \texttt{type}, does not have \texttt{LEX} in its domain). One might object to the introduction of a whole new notation just to reduce the original three rules to two (plus two metarules), but I will show below that the machinery pays off in many more cases.

Although I will not go into the details of lexical fillers here (see section 3.3), this is a good place to say something about the fillers of the tense operators. \texttt{V_{1}} determines whether the finite verb has present or past tense by means of the abstract fillers \texttt{PRES} or \texttt{PAST} respectively. In Seuren’s formalization, the two fillers of \texttt{V_{2}} are \texttt{∅} or a perfective auxiliary verb (\emph{have} in English, \emph{hebben/zijn} in Dutch, and \emph{haben/sein} in German). However, in our view, it is more consistent to use abstract fillers for this operator as well. In fact, this is exactly what Seuren (1996, p.84) assumes at a more theoretical level: “For the European languages the two possible values for \texttt{V_{2}} are \texttt{SIMULTANEOUS} and \texttt{PRECEinding.” Looking at it from a technolinguistic point of view, we have the implementation deviate from the formalization in this respect and bring it in accordance with the underlying theory: we use the abstract fillers \texttt{SIM} and \texttt{PDEC} for \texttt{V_{2}}. This is not only more consistent with the treatment of \texttt{V_{1}}, it also brings the rules for English, Dutch and German in line with each other as the abstract fillers are not language-specific.

### 3.2.2 Lexical argument frame

The lexical argument frame is where the arguments of the lexical predicate come into being. As described in chapter 2, both the formation rules and the lexical argument frame fit a general SA-format. This format states that every S is expanded into a predicate V and a maximum of three arguments. If present, the first argument (NP
or S) is the subject; if there are two or more arguments, the last one (NP or S) is the direct object; if there are three arguments, the middle one (NP) is the indirect object (see Figure 3.3). Seuren (1996, p.25) presents the SA-format as a threefold rewrite rule:

\[
S'' \\
  V_{t1} \quad S' \\
  V_{t2} \quad S^0 \\
V_{LEX} \quad NP/S \quad NP \quad NP/S
\]

subject \quad indirect \quad direct \quad object

Figure 3.3 Slots in the lexical argument frame.

\[
V + (NP/S) \\
S \rightarrow \quad V + NP/S + NP/S \\
V + NP/S + NP + NP/S
\]

The examples in (7) illustrate that a subject or object S may have either zero (7a), one (7b) or two (7c) tenses, depending on the higher predicate.

(7) \hspace{1cm} a \hspace{1cm} I \hspace{0.1cm} helped \hspace{0.1cm} John \hspace{0.1cm} (to) \hspace{0.1cm} sleep. \\
\hspace{1cm} b \hspace{1cm} I \hspace{0.1cm} expect \hspace{0.1cm} John \hspace{0.1cm} to \hspace{0.1cm} have \hspace{0.1cm} slept. \\
\hspace{1cm} c \hspace{1cm} I \hspace{0.1cm} knew \hspace{0.1cm} that \hspace{0.1cm} John \hspace{0.1cm} slept.

However, the threefold SA-format is merely a pattern specification, which we do not find explicitly in Seuren’s formation rules. As mentioned in the previous section, there is one general formation rule introducing the lexical predicate and its frame in a rather abstract manner:

\[
S^0 \rightarrow V_{t1}X + <\text{lex.arg.frame}>
\]

The element \(<\text{lex.arg.frame}>\) refers to a part of the lexical specification (see also section 3.3), which states for each predicate the obligatory and/or optional arguments. Obviously, such a referring element cannot be mapped directly onto an EAG formulation in the implementation.

We encode the lexical information about the argument frame in three separate V affixes, with non-terminals su (subject), optio (optional indirect object) and optdo
(optional direct object) respectively.\textsuperscript{12} The instantiation of these affixes comes from the lexicon and is copied to three dummy elements $\text{compl}<\text{su}/\text{optio}/\text{optdo}>0$, which are to be rewritten into the appropriate elements.\textsuperscript{13} The relevant EAG rule is extended as follows:

\[
S<\text{LEX}>: \quad V<\text{LEX}, \text{su}, \text{optio}, \text{optdo}>0,
\text{compl}<\text{su}>0, \text{compl}<\text{optio}>0, \text{compl}<\text{optdo}>0.
\]

Of course, now we also need metarules for the affix non-terminals and rewrite rules for the dummy element. The non-terminals $\text{optio}$ and $\text{optdo}$ are either empty, in which case there is no indirect object or direct object present (the dummy element $\text{compl}<0$ is rewritten to empty and thus disappears), or they are replaced by non-terminals $\text{io}$ and $\text{do}$, respectively. To express Seuren's general SA-format, we split the non-terminals $\text{su}$ and $\text{do}$ up in two further non-terminals $\text{np}$ and $\text{s}$; the non-terminal $\text{io}$ has the only option $\text{np}$:

\[
\begin{align*}
\text{optio}: & \quad \text{io}; . \\
\text{optdo}: & \quad \text{do}; . \\
\text{su}: & \quad \text{np}; \text{s}. \\
\text{do}: & \quad \text{np}; \text{s}. \\
\text{io}: & \quad \text{np}.
\end{align*}
\]

$\text{compl}<0:$ .

The non-terminal $\text{np}$ is instantiated as $\text{NP}$ or $\text{N PX}$. The hypergrammar rewrites $\text{compl}<\text{NP}>0$ as a normal $\text{NP}$ and $\text{compl}<\text{NPX}>0$ as an $\text{NP}$ with co-reference feature $X$ (see section 3.2.9 for further details on noun phrases):

\[
\begin{align*}
\text{np}: & \quad \text{NP}; \text{NPX}. \\
\text{compl}<\text{NP}>0: & \quad \text{NP}<>. \\
\text{compl}<\text{NPX}>0: & \quad \text{NP}<\text{X}>.
\end{align*}
\]

The non-terminal $\text{s}$ results in a subject or object $S$ with zero, one or two tenses, via the instantiations $\text{LEX}$, $\text{PERF}$ and $\text{FIN}$, respectively:

\[
\begin{align*}
\text{type}: & \quad [\text{FIN}] \text{FIN} \ll \\
& \quad [\text{PERF}] \text{PERF} \ll \\
& \quad [\text{LEX}] \text{LEX}.
\end{align*}
\]

$\text{s}: \quad \text{FIN}; \text{PERF}; \text{LEX}$.

$\text{compl}<\text{FIN}>0: \quad \text{S<! [FIN]type>}$.

$\text{compl}<\text{PERF}>0: \quad \text{S<! [PERF]type>}$.

$\text{compl}<\text{LEX}>0: \quad \text{S<! [LEX]type>}$.

\textsuperscript{12} In the implementation we take the subject to be an obligatory element rather than optional, since all predicates accounted for in the lexicon do indeed have at least one argument. An example of a predicate without an argument would be the verb $\text{rain}$, which receives a dummy subject $\text{it}$.

\textsuperscript{13} The 0 (zero) attached to the dummy element is not standard EAG notation (see the appendix). It ensures that the auxiliary $\text{compl}$-node will not appear in the resulting structure itself, but only its daughter will (if it has one).
The exclamation mark refers to the hierarchy mechanism. It indicates that a new substitution process must begin, at the entry point indicated between square brackets (see also the appendix). These entry points are marked as such in the metarule for type.

Note that this implementation misses a theoretical generalization made by Seuren about the SA-format, namely that a direct object can only be present if there also is a subject, and that an indirect object can only be present if there also is a direct object. In fact, what we have implemented is

\[ S \rightarrow V + NP/S (+ NP) (+ NP/S) \]

instead of the much stricter constraint

\[ S \rightarrow V (+ NP/S ((+ NP) + NP/S)) \]

which is a shorter version of Seuren's threefold pattern specification. Although it is possible to express the stricter constraint in GRAMGEN, we have decided not to do so because it would complicate the grammar (and the choices to be made by the user) considerably and because the constraint is not expressed in the formalization either.\footnote{An additional ground for not implementing the stricter constraint is that it seems to be too strict for some predicates. Our implemented version acknowledges the fact that there are predicates with argument frames that deviate from the general pattern, such as, for example, help in English, gehoorzamen in Dutch and glauben in German (see sections 4.2.4 and 5.5.1).}

### 3.2.3 Modality and future tense

Seuren (1996) argues for a position in between the two tense operators, primarily for the expression of future tense and secondarily also for (other) modals. In English, this position can be filled by any of the modal verbs with an incomplete paradigm: will, may, can et cetera. Since these verbs have present and past forms, as illustrated in (8a,b), they must be inside the scope of the first tense operator. Placing them above the second tense operator, which is responsible for perfect tense, guarantees that they have no perfect tense themselves (8c), but do allow a perfect form of the main verb below them (8d). Their position between the tenses also ensures that they have no infinitives (8e).
(8)  
  a. John will sleep.  
  b. John would sleep.  
  c. * John has would sleep.  
  d. John will have slept.  
  e. * John hoped to will sleep.  

Seuren also argues that in German, the position is reserved for the future auxiliary *werden*, which has the same paradigm (9). In Dutch the position is empty: in this language future tense as well as modalities are expressed by main verbs with full paradigms, selecting clausal arguments (10).  

(9)  
  a. Jan wird schlafen.  
  b. Jan wurde schlafen.  
  c. * Jan hat schlafen werden.  
  d. Jan wird geschlafen haben.  
  e. * Jan hoffte schlafen zu werden.  

(10)  
  a. Jan zal slapen.  
  b. Jan zou slapen.  
  c. Jan heeft zullen slapen.  
  d. Jan zal hebben geslapen.  
  e. Jan hoopte te zullen slapen.  

It is clear that the patterns are essentially the same for the English modals and the German future auxiliary (see also Figure 3.4). The relevant formation rules are:

<table>
<thead>
<tr>
<th>English</th>
<th>German</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S^n$ $\rightarrow$ $V_{nt} + S^M / S'$</td>
<td>$S^n$ $\rightarrow$ $V_{nt} + S^\text{FUT} / S'$</td>
</tr>
<tr>
<td>$S^M$ $\rightarrow$ $V_M + S'$</td>
<td>$S^\text{FUT}$ $\rightarrow$ $V_{\text{FUT}} + S'$</td>
</tr>
</tbody>
</table>

The modality or future tense operator has a somewhat different status from the tense operators and the lexical predicate we have seen above. It is not an obligatory element in the SA: rather, it may or may not be plugged into the general tense scheme. Seuren expresses this by using a second superscript for the mother S, following the tense superscript. In the EAG, however, we use the same affix position so as to fit the operator into the same hierarchy. Now, all we have to do to accommodate for this optional element is integrate it into the metarule for tense and the type hierarchy, with the rest of the grammar staying the same.

---

15 I assume that, if a past participle form of *will* (still) existed, it would be *would*. This is likely on historical grounds, see Visser (1973, pp. 2221-2222): “The past participles of the verbs will, can, and may [...] were formerly used in the modal pluperfect in such clusters as ‘had would’, ‘had could’ and ‘had might’.”  
16 But see section 3.2.4 on the so-called *middle auxiliaries*, which will be treated as a special type of modals, in English as well as in Dutch.
Figure 3.4 The modal future tense position in between the tenses

English:

tense:: ...; MODAL.
type:: [FIN] FIN << (MODAL) << [PERF] PERF << ...  

Note that MODAL needs to precede the entry point [PERF] because there can be no modal in an embedded perfective S, as illustrated in (11).

(11)  *  I expect John to have slept.

For German, we could implement Seuren’s formalization in a similar way.

German:

tense:: ...; FUT.
type:: [FIN] FIN << (FUT) << [PERF] PERF << ...

However, Richter (2000) argues that in German not only the future auxiliary werden occupies this position in between the two tenses, but also the modals versprechen and pflegen do. These verbs (in German traditionally called Halbmodalen) show the same defective paradigm as the English modals, as illustrated for pflegen in (12). 17

(12)  
  a  Jan pflegt zu schlafen.
  b  Jan pflegte zu schlafen.
  c  *  Jan hat zu schlafen pflegen/gepflegt.
  d  Jan pflegt geschlafen zu haben.
  e  *  Jan hoffte zu schlafen zu pflegen.

---

17 Two other verbs that belong to the class of Halbmodalen are scheinen and drohen. Although these verbs share the same paradigm as shown for pflegen in (12), the fact that they additionally allow finite clauses below them is reason for Richter to analyse them differently.
Following Richter in our implementation, we can nicely dispense with the difference between German and English altogether by using the affix terminal MODAL instead of FUT for werden and the (semi-)modals.\footnote{An argument against such a treatment of these verbs would be that they do not have bare infinitives below them, but \texttt{z(to)}-infinitives. This complication can be dealt with in the transformational grammar, however (see section 5.5.5; see also Richter 2000, p.187-191).}

\textit{English and German:}

\begin{verbatim}
tense:: \ldots MODAL.
type:: [FIN] FIN << (MODAL) << 
       [PERF] PERF << \ldots
\end{verbatim}

Incidentally, it should be noted that similar observations seem to hold for Dutch \textit{lijken} and \textit{schijnen}. Also, \textit{zullen} in its potentialis use (as opposed to its primary use for future tense) seems to behave like a real modal. However, since it is not our primary aim to expand the coverage of Semantic Syntax, we have not investigated this matter any further. Hence, our implementation will not deviate from Seuren’s formalization in this respect.

### 3.2.4 Middle auxiliaries

Seuren (1996, p.137-141 and p.221-222) recognizes a special construction in English and Dutch with a typical defective paradigm. It concerns constructions with \textit{be to} and \textit{be going to} in English, and \textit{vallen te} in Dutch (meaning ‘be humanly possible’ or ‘be feasible’; literally ‘fall to’). Seuren calls these verbs middle auxiliaries. Their paradigms are illustrated in (13) and (14), respectively.

\begin{verbatim}
(13) a  This is (going) to be done.
b  This was (going) to be done.
c *  This has been (going) to be done.
d *  This had been (going) to be done.
e *  I expect this to be (going) to be done.
f *  This is (going) to have been done.
g *  This will be (going) to be done.
\end{verbatim}

\begin{verbatim}
(14) a  Dit valt te doen.
b  Dit viel te doen.
c *  Dit is te doen gevallen.
d *  Dit was te doen gevallen.
e *  Dit moet te doen vallen.
f *  Dit valt gedaan te zijn.
g *  Dit zal te doen vallen.
\end{verbatim}

As Seuren (1996, p.138) puts it, these constructions “show a defective paradigm which is reminiscent of, though not identical to, the modals”. Like the modals, mid-
dle auxiliaries occur in simple present and past tense (the a and b examples), but not in the perfect tense (c and d) nor as infinitives in embedded clauses (e). Unlike the modals, however, they do not allow for a perfective complement clause either (f). And finally, (13g) shows that in English these auxiliaries may not be placed under a modal themselves. The Dutch example (14g) is equally ungrammatical, but possibly for a different reason: remember that the Dutch “modals” are treated as lexical verbs with an S-complement (so (14g) is in fact completely parallel to (14e); but recall my remark about Dutch *liken*, *scheinen* and *zullen* at the end of the previous section).

Seuren accounts for these facts by means of a special S-type which jumps directly from S" to S⁰, bypassing S' (see Figure 3.5). The following formation rules express this situation:

![Diagram of formation rules for middle auxiliaries under a special S, bypassing S'.]

**Figure 3.5 The middle auxiliary under a special S, bypassing S'**

### English

\[
\begin{align*}
S'' & \rightarrow V_{tl} + S^M / S' / S^{0\text{MAUX}} \\
S^{0\text{MAUX}} & \rightarrow V_{\text{Maux}} + S^0
\end{align*}
\]

### Dutch

\[
\begin{align*}
S'' & \rightarrow V_{tl} + S' / S^{0\text{MAUX}} \\
S^{0\text{MAUX}} & \rightarrow V_{\text{Maux}} + S^0
\end{align*}
\]

However, when we want to integrate Seuren’s treatment of these constructions into the implemented hierarchy we run into difficulties, since we cannot easily say something like “if MAUX is chosen, then skip the otherwise obligatory PERF”. The only implementation that comes close to this would involve a disjunction of MAUX and PERF.

### English:

\[
\begin{align*}
\text{maux/perf:: MAUX}; \text{PERF}. \\
\text{type:: } [\text{FIN}] \text{ FIN} & \ll \text{MODAL} \ll \\
& \ll \text{PERF} \text{ maux/perf} \ll \ldots
\end{align*}
\]

But this raises two major problems. First, the position of MAUX after the entry point [PERF] allows middle auxiliaries in S'-complements, whereas examples (13e) and (14e) show that this is unacceptable: a middle auxiliary can never occur as an infinitive in an embedded clause. Second, in the English grammar, this hierarchy has no
means to exclude the co-occurrence of a modal and a middle auxiliary, which is also unacceptable given (13g).

Let us, therefore, pursue another implementation, which solves both problems at once and at the same time seems to be more in line with Seuren’s remark about the parallels between middle auxiliaries and modals. That is, let us treat the middle auxiliaries as a special type of modals. This leaves us with the question how to ensure the absence of a perfective tense in these constructions. The answer to that lies in a different approach to this absence: we do not regard it as the second tense operator being skipped altogether, but as the operator having SIM for its obligatory filler (recall that this is the filler that results in a simple present or past tense, whereas PREC leads to a perfective tense – see p.32).

This can be accomplished by means of a second affix with S, a selectional affix. We will reserve this affix in general for transmitting restrictive information down to a lower level in the tree that is being generated.

**English and Dutch:**

\[
S<\text{MODAL},> : \quad V<\text{MODAL},\text{perf},>, S<\text{type},\text{perf}>.
\]

The choice of a middle auxiliary (which is represented in the lexicon as \(V<\text{MODAL},\text{NPERF}>\), i.e. as a special type of modal) will introduce the affix terminal \(\text{NPERF}\) on its sister \(S\), through unification of the two instances of \(\text{perf}\). We furthermore split up the hyperrule for \(S<\text{tense}>\), to pass on the affix until the perfective tense operator is reached. It limits the choice of lexical fillers for this operator to \(\text{SIM}\).

**English and Dutch:**

\[
\text{type}: [\text{FIN}] \text{FIN} << (\text{MODAL}) << \text{PERF} \text{PERF} << \ldots
\]
\[
\text{tense}: \quad \text{FIN}.
\]
\[
\text{perf}: \quad \text{NPERF} ; .
\]

\[
S<\text{tense},\text{perf}> : \quad \ldots, S<\text{type},\text{perf}>.
\]
\[
S<\text{PERF},> : \quad V<\text{PERF},,,\text{perf},>, S<\text{type},>.
\]
\[
S<\text{PERF},\text{NPERF}> : \quad V<\text{PERF},,,\text{NPERF}>, S<\text{type},>.
\]

Note that we implement this exactly the same for English and for Dutch, which means we assume Dutch also to have a modal position after all.

Our solution introduces an important generalization, by treating modals and middle auxiliaries on a par. The machinery we need for the implementation may seem a little ad hoc at this stage. However, I will show below that we need the selectional affix with \(S\) in a couple of other cases, for the same purpose of passing restrictions down to a lower tree level (see, for example, section 3.2.6). We even need the particular distinction between a regular \(S<\text{PERF}>\) (with tense operator \(\text{PREC}\) or \(\text{SIM}\)) and an \(S<\text{PERF},\text{NPERF}>\) (with only \(\text{SIM}\)) for certain restricted complement clauses, as will become clear in the next section.
3.2.5 Adverbials

Seuren (1996, pp.116-128) has put quite some effort into figuring out the distribution facts of adverbials. Adverbs and prepositional phrases are inserted into the SA at various levels, by various formation rules. The system is a little more complex for English than for the other two languages, but the overall mechanism is similar for all three of them:

**English**

\[
\begin{align*}
S^0_{ADV0} & \rightarrow V_{Adv0} + S^0_{ADV1} / S^0_{ADV2} / S^0 \\
\text{or: } V_{Prep0} + S^0_{ADV1} / S^0_{ADV2} / S^0 + NP \\
S^0_{ADV1} & \rightarrow V_{Adv1} + S^0_{ADV2} / S^0 \\
\text{or: } V_{Prep1} + S^0_{ADV2} / S^0 + NP \\
S^0_{ADV2} & \rightarrow V_{Adv2} + S^0_{ADV2} / S^0 \\
\text{or: } V_{Prep2} + S^0_{ADV2} / S^0 + NP \\
S^M & \rightarrow V_M + S^0_{ADV2} / S^0 \\
S^0_{ADV2} & \rightarrow V_{Adv2} + S^0_{ADV2} / S^0 \\
\text{or: } V_{Prep2} + S^0_{ADV2} / S^0 + NP \\
S^0 & \rightarrow V_{Adv3} + S^0_{ADV3} / S^0 \\
\text{or: } V_{Prep3} + S^0_{ADV3} / S^0 + NP
\end{align*}
\]

**Dutch and German**

\[
\begin{align*}
S^0_{ADV0} & \rightarrow V_{Adv0} + S^0_{ADV1} / S^0 \\
\text{or: } V_{Prep0} + S^0_{ADV1} / S^0 + NP \\
S^0_{ADV1} & \rightarrow V_{Adv1} + S^0_{ADV1} / S^0 \\
\text{or: } V_{Prep1} + S^0_{ADV1} / S^0 + NP \\
S^0_{ADV2} & \rightarrow V_{Adv2} + S^0_{ADV2} / S^0 \\
\text{or: } V_{Prep2} + S^0_{ADV2} / S^0 + NP \\
S^0 & \rightarrow V_{Adv3} + S^0_{ADV3} / S^0 \\
\text{or: } V_{Prep3} + S^0_{ADV3} / S^0 + NP
\end{align*}
\]

Level 0 contains the highest possible element in a sentence, which can occur only once. These elements are the so-called super-adverbials like moreover and in that case, and also conjunctions like so and but. In English, there are two adverbial levels directly below the super-adverbial: one for adverbials that occur only once (level 1) and one where adverbials can be inserted recursively (level 2). Level 1 adverbs are, for example, fortunately and allegedly; at level 2 we find adverbials of place and time, and also the negation not. Level 1 is above the first tense operator; level 2 is situated either above the first tense operator ($S^0_{ADV2}$) or below the operator and the modal auxiliary ($S^0_{ADV2}$). In Dutch and German we have a more clear-cut distinction: level 1 is for adverbials that occur (recursively) above the first tense operator, whereas level 2 houses the adverbials occurring (also recursively) below the operator. All of the adverbials at these levels are traditionally called sentence adverbials. The lowest adverbial level in the SA is level 3. This level accommodates, among others, the manner adverbials: fast, carefully, well et cetera. The example in (15) illustrates how all four adverbial levels may be combined.\(^{10}\)

(15) [So]$_0$ [unfortunately], John did [not]$_2$ sleep [well].

---

\(^{10}\) Note that this particular example was carefully chosen to illustrate the hierarchical order of the adverbial levels. The hierarchy in the SA-structure is, however, not always reflected in the surfacing linear order of the adverbs, as various transformations may affect their positions; see also sections 4.3.2 and 4.4.3.
The adverbial scheme as given by Seuren is illustrated in Figure 3.6, where dotted lines indicate optional elements, dashed lines indicate possible recursion, and the shaded areas only apply to the English SA.

As can be seen, adverbs and adverbial prepositions behave in the same way. The sole difference is that prepositions not only have a subject S, but also an object NP. Besides that, they have exactly the same SA distribution as the adverbs: they can occur at the same levels, below the same mother nodes. If we had another mechanism to ensure that prepositions have an NP object and adverbs do not, it would be unnecessary to make the distinction at the SA-level. The distinction will only become relevant at a later stage, when the surface category comes into play, but this information is stored separately in the lexicon anyhow (see also sections 3.3 and 4.3.4).

Since we do in fact have an independently motivated mechanism for selecting objects when necessary (see section 3.2.2), we decide to drop the distinction indeed. Thus, we can stick to the same strategy as we have done before: mother S and head-daughter V always have the same affix.
tense:: ...; ADV0; ADV1; ADV2; ADV3.

Now, the adverbial scheme can easily be incorporated in the hierarchy. For English, the metarules should be extended as follows (note that the optionality brackets are nested; V<ADV2> may only occur below V<FIN> if there is a V<MODAL> in between them).

English:
type:: (ADV0) << (ADV1) << (ADV2*) << FIN << (MODAL << (ADV2*)) << PERF << (ADV3*) << LEX.

The question is where to situate in this hierarchy the entry points for embedded clauses relative to the adverbial levels. According to Seuren (1996, p.89), a finite complement-S may be accompanied by a level 2 adverbial (16a) but not by a level 1 adverbial (16b) or a level 0 adverbial (16c), so the entry point [FIN] should be placed between ADV1 and ADV2.

(16) a I knew that John did not sleep.
    b * I knew that unfortunately John slept.
    c * I knew that so John slept.

It is harder to establish where the entry point [PERF] should be positioned though. It has already been shown that [PERF] needs to follow MODAL, because there can be no modal in a perfective complement-S. However, it is also obvious that such an S can contain a level 2 adverbial (17); again, see Seuren (1996, p.89). This means that the entry point should be placed in between MODAL and ADV2. But in the given hierarchy that would be computationally undesirable because the nested brackets precisely state that ADV2 is only present if MODAL is too.

(17) I expect John not to have slept.

This problem leads us to reconsider the splitting up of adverbial level 2 into a pre-tense-operator level and a post-tense-operator level, the latter depending on the presence of a modal. Why did Seuren create this distinction in the first place? The primary reason seems to be that when a modal is present, it is semantically relevant whether it is inside or outside the scope of the adverb (below or above it). So, rather than a pre- and post-tense-operator level 2, there appears to be a pre- and post-modal level 2. This means we can reformulate the hierarchical order

(ADV2*) << FIN << (MODAL << (ADV2*)) << PERF

as

FIN << ((ADV2*) << MODAL) << (ADV2*) << PERF.

The effect is the same: only when a modal is present, there are two separate levels 2. Now, there is no computational objection to place the entry point [PERF] in between MODAL and (the second) ADV2, as they may occur independently. We do need to make some adjustments in the transformational grammar, however, to deal with the new situation that the two tense operators can be separated by an adverbial.
These adjustments, pertaining to AUX-delimitation and Adverb Placement, will be explained in sections 4.4.1 and 4.4.3.

The way we came upon this problem and the way we tackled it illustrate the merits of our technolinguistic approach. The implementation brought to light something undesirable that lay hidden in the formalization. It led us to reconsider the theoretical backgrounds, for which we found another, more suitable implementation.\(^{20}\) Note that this new implementation also brings the English grammar more in line with those for Dutch and German: in those languages, level 2 was already exclusively situated below the first tense operator.\(^{21}\)

The entry point [LEX] for tenseless S-complements should precede ADV3, as sentences like (18) are fully grammatical.

(18) I helped John (to) sleep well.

\textit{English:}

\begin{verbatim}
  type:: (ADV0) << (ADV1) <<
  [FIN] FIN << ((ADV2*) << MODAL) <<
  [PERF] (ADV2*) << PERF <<
  [LEX] (ADV3*) << LEX.
\end{verbatim}

For Dutch and German, the same reasoning holds. Since we have assumed a modal position in German (following Richter 2000) and in Dutch (for the middle auxiliary) too, we decide to split up adverbial level 2 for these languages as well, for the same reasons of scope. A difference with English is that the German and Dutch entry points [FIN] must precede ADV1, because a finite complement-S does allow a level 1 adverbial. The other two entry points have the same positions as in English: before ADV2 and before ADV3, respectively.

\textit{Dutch and German:}

\begin{verbatim}
  type:: (ADV0) <<
  [FIN] (ADV1*) << FIN << ((ADV2*) << MODAL) <<
  [PERF] (ADV2*) << PERF <<
  [LEX] (ADV3*) << LEX.
\end{verbatim}

For all three languages, the hyperrule only needs to be extended to accommodate the NP argument of prepositional phrases. As described earlier, we already have a mechanism for this which was needed for the lexical argument frame: we use the dummy argument \texttt{compl<optdo>0} which either expands to an NP or to nothing.

\(^{20}\) Applying the equivalent changes to Seuren's formalization would result in more complex formation rules, as we would need to distinct the two types of adverbial levels. Seuren (personal communication) suggests to remove the adverbials as such from the formation rules - thereby making them a lot easier to read - and instead formulate an independent principle: "For each S, there is optional adjoining of members of Adv' above S" (see also Seuren 1999).

\(^{21}\) In German and Dutch the adverbial level in between the two tense operators has not been causing problems because the cyclic treatment placed the adverbs in question outside the V-cluster, as opposed to the cyclic treatment of the English level 2 adverbs. As Seuren (personal communication) points out, the grammar for French did contain a serious problem in this respect. Careful scrutiny of the French grammar revealed not just a computationally undesirable situation, but an actual descriptive inadequacy. The same adjustments solve this problem too, however.
depending on the instantiation of the affix optdo. This instantiation is inherited from lexical information about V: adverbs (as well as the other predicates without object) have a zero affix terminal, and prepositions have the affix terminal NP.

\[ S\langle \text{tense, perf} \rangle: \quad V\langle \text{tense, , optdo, >, ..., compl} \langle \text{optdo} \rangle, 0. \] 

A final remark should be made here in relation to the entry point [PERF]. Some complement clauses allow for a level 2 adverbial, although they do not allow for a perfect tense (see Seuren 1996, p.89). This occurs in all three languages under consideration, as illustrated in (19), (20), and (21).

(19)  
  a  I told him not to go.  
  b  * I told him to have gone.  
(20)  
  a  Ich bat ihn, nicht zu gehen.  
  b  * Ich bat ihn, gegangen zu sein.  
(21)  
  a  Ik verzocht hem niet te gaan.  
  b  * Ik verzocht hem te zijn gegaan.  

Seuren describes these clauses as S'-embeddings with the obligatory selection of SIM for the tense operator. For these cases, we employ the same mechanism as was needed before for the middle auxiliaries. The selection of SIM is again enforced by the affix terminal NOPERF, which is now introduced via the lexical argument frame rather than by the middle auxiliary:

\[ \text{compl} \langle \text{NOPERF} \rangle, 0: \quad S \langle ! \left[ \text{PERF} \langle \text{type} \rangle, \text{NOPERF} \rangle \right. \rangle. \] 

3.2.6 Topicalization and questions

Seuren (1996, p.104) emphasizes that the syntactic phenomena of topicalization or fronting as in (22a) and question formation as in (22b) are very similar. He accounts for this similarity by means of an attraction mechanism adhering to an operator V_A, with the abstract filler Foc or Que. In section 4.4, I will come back to the attraction mechanism, which is part of the postcyclic rules; right now, the focus is on the role of the operator in the formation rules. As Figure 3.7 illustrates, the attraction operator is a high operator generated at the top of the SA tree. This goes for English, Dutch and German.

(22)  
  a  In that way I can help you.  
  b  Who can help you?  

The formation rules define how the attraction operator interacts with the higher adverbial levels. For English, the relevant rules are:  

22 Marginally, a verb like Dutch verzoeken does seem to allow an embedded perfect tense, as in Ik verzocht hem het artikel morgen geön geön te hebben ("I requested him to have read the article by tomorrow"). However, this is not accounted for in the formalization, nor in the implementation.
Figure 3.7 The attraction operator.

\[ S^{n\text{ADV0}} \rightarrow V_{\text{Adv0}} + S^{n\text{ATTR}} / S^{n\text{ADV1}} / S^n \]
\[ \text{or: } V_{\text{Prep0}} + S^{n\text{ATTR}} / S^{n\text{ADV1}} / S^n + \text{NP} \]
\[ S^{n\text{ATTR}} \rightarrow V_{\text{Attr}} + S^n \]
\[ S^{n\text{ADV1}} \rightarrow V_{\text{Adv1}} + S^n \]
\[ \text{or: } V_{\text{Prep1}} + S^n + \text{NP} \]

If a level 0 adverbial is present, it precedes the attraction operator. A level 1 adverbial cannot be combined with the attraction operator: only one of the two may be present.

The situation in German and Dutch is similar, though not exactly the same:

\[ S^{n\text{ADV0}} \rightarrow V_{\text{Adv0}} + S^{n\text{ATTR}} / S^{n\text{ADV1}} / S^n \]
\[ \text{or: } V_{\text{Prep0}} + S^{n\text{ATTR}} / S^{n\text{ADV1}} / S^n + \text{NP} \]
\[ S^{n\text{ATTR}} \rightarrow V_{\text{Attr}} + S^{n\text{ADV1}} / S^n \]
\[ S^{n\text{ADV1}} \rightarrow V_{\text{Adv1}} + S^{n\text{ADV1}} / S^n \]
\[ \text{or: } V_{\text{Prep1}} + S^{n\text{ADV1}} / S^n + \text{NP} \]

Again, a level 0 adverbial always precedes the attraction operator. But in German and Dutch, contrary to the English situation, a level 1 adverbial can be combined with the attraction operator. If so, the adverbial follows the operator.

The difference between English on the one hand and German and Dutch on the other is easy to understand if we look upon the attraction operator as a special case of the level 1 adverbials. Recall that in English this adverbial level is non-recursive, whereas in German and Dutch it is a recursive level (see the previous section). In

\[ ^{23} \text{I leave out } S^{n\text{ADV2}} \text{ from Seuren's (1996, p.92) original formation rules, since we have established there should not be a pre-tense-operator adverbial level 2 (see section 3.2.5, but see also footnote 20).} \]
that light, if the operator is considered a level 1 adverbial, it is only natural that in English the operator cannot be combined with “another” level 1 adverbial - just as we cannot combine two regular level 1 adverbials. For the same reasons, the German or Dutch attraction operator does co-occur with a level 1 adverbial. What sets the operator apart from the “other” level 1 adverbials in these two languages, is that it always comes first and that it may only occur once itself.\footnote{If it were not for these last two differences, the otherwise completely identical distribution facts would justify a formalization and implementation indeed treating the attraction operator as a level 1 adverbial. Although the differences only hold for German and Dutch, we have decided to follow Seuren in maintaining a distinct treatment of the English operator as well, so as to keep the three grammars in line as much as possible.}

The integration of the attraction operator in the metagrammar is quite uncomplicated:

\begin{verbatim}
tense:: ...; ATTR.

English:
type::   (ADV0) << (attr/advl) <<
         [FIN] FIN << ...
attr/advl:: ATTR; ADV1.

Dutch and German:
type::   (ADV0) << (ATTR) <<
         [FIN] (ADV1+) << FIN << ...
\end{verbatim}

The position of the entry point [FIN] after ATTR ensures that in all three languages the attraction operator normally only occurs in main clauses (see Seuren 1996, p.106). The only exception to this is found with some verbs, such as tell, know, or ask, which may take a dependent question as object clause. Seuren accounts for this in the lexical argument frame of such verbs, by specifying $\text{NP}[\text{Que}+S'']$ as one of the direct object alternatives.

Let us take a closer look at that specification. Taken literally, it would correspond to an SA structure as in Figure 3.8.a, which seems to be rather inconsistent with the rest of the formalization though. First of all, a filler like Que should not appear without a category label (see also Seuren 1996, p.46). Since we are dealing here with the same attraction operator as in main clauses, we would expect this Que to have the label $V_{\text{Adv}}$ as well. Second, it seems awkward to have an NP with a finite S as one of its two dependent nodes. What we do find regularly in the rest of the grammar(s) - although I did not go into that yet; see section 3.2.9 on Noun Phrases - is a construction where the complement-S is headed by an NP-node; in that case, S is always the only daughter of NP. Therefore, we conclude that an SA structure as in Figure 3.8.b is more in line with the rest of the grammar (and is probably also the structure that Seuren envisaged). It corresponds to a much more complex specification: $\text{NP}[s^{\text{ATTR}}[\text{VAdv}[\text{Que}]+S'']]$. 

\begin{verbatim}
\text{NP}[s^{\text{ATTR}}[\text{VAdv}[\text{Que}]+S'']]
This leaves us with the question how to implement the generation of such SA structures. A possible solution presents itself quite naturally: we can introduce an extra entry point \([\text{QUE}]\) to which we refer in the lexical argument frame (via the intermediate NP-node) and which we place before the attraction operator in the hierarchy.

\[
\begin{align*}
\text{s::} & \quad \ldots ; \text{QUE}. \\
\text{compl<QUE>0:} & \quad \text{NP<QUE>}. \\
\text{NP<QUE>:} & \quad \text{S<! [QUE] type,>}. \\
\end{align*}
\]

**English:**

\[
\begin{align*}
\text{type::} & \quad (\text{ADV0}) \ll \\
& \quad [\text{QUE}] \ (\text{attr/adv1}) \ll \\
& \quad [\text{FIN}] \ \text{FIN} \ll \ldots
\end{align*}
\]

**Dutch and German:**

\[
\begin{align*}
\text{type::} & \quad (\text{ADV0}) \ll \\
& \quad [\text{QUE}] \ (\text{ATTR}) \ll \\
& \quad [\text{FIN}] \ (\text{ADV1*}) \ll \text{FIN} \ll \ldots
\end{align*}
\]

However, three complications emerge. One is how to force the choice of the right filler for the attraction operator, i.e. \(\text{Que}\) instead of \(\text{Foc}\). The second complication is how we should exclude the generation of a level 1 adverbial in English, rather than the attraction operator. A third complication is that this implementation predicts that all verbs that allow for an embedded question also allow for an embedded finite clause, since the attraction operator is an optional element in the hierarchy. That this is not true for all verbs is shown in (23).

(23) * I ask that you helped me.

Although these complications can be fixed to a certain extent within the hierarchy, they cannot be eliminated entirely and thus encourage us to look for a different strategy. Why not separate the embedded question operator from the hierarchy altogether? Instead of using a special entry point \([\text{QUE}]\) we could generate the
question operator directly, followed by an S-complement referring to entry point [FIN]:

\[ NP<\text{QUE}>: \quad \text{v<ATTR, , , QUE>}, \quad S<! [FIN] type, >. \]

It must be specified in the lexicon that the filler of \( \text{v<ATTR, QUE>} \) is \text{Que} (see also section 3.3). An independent reason for separating the embedded question operator from the hierarchy would come into play if we wanted to expand the grammar to generate sentences like (24) with the operator in an infinitival complement clause. I will not pursue this here, since it is not incorporated in Seuren’s formalization, but it is clear that this type of complementation would involve the same question operator, but now followed by an \( S<! [LEX] type> \).

(24) I don’t know who to believe.

Only one small problem attached to this implementation needs to be solved. The V- and S-nodes will not share a mother S, but will come directly below the NP-node (as in Figure 3.8.a). Therefore, we introduce an intermediate S and make use of the sectional affix, which was introduced earlier in the context of middle auxiliaries (see p.38-41):

\[ NP<\text{QUE}>: \quad S<\text{ATTR, QUE}>. \]
\[ S<\text{ATTR, QUE}>: \quad \text{v<ATTR, , , QUE>}, \quad S<! [FIN] type, >. \]

A completely different question is how to generate the elements that are to be attracted by the operator in the transformational grammar. As said at the beginning of this section, I will address the attraction mechanism itself later on (see section 4.4.4), but what needs to be solved in this module of the system is the following. The attraction mechanism is triggered by the operator Que or Foc, but it applies to another element in the SA-structure: something marked [+WH] or [+Foc], respectively. How do we make sure such an element is indeed generated in the SA? This is a well-known problem, which has also played a role, for example, in relation to quantifiers in the formalization of logic languages.

Seuren (1996, p.110, 148) has not integrated this into the formalization: “the formal introduction of what has not been accounted for by the Formation Rules” and “we do not have a formal structural account of the origin of the feature [+Foc]. It is assumed that external factors, to do with discourse and situation, are responsible for the feature.” Such contextual factors can also not be accounted for in the implementation as yet, since the generator only produces isolated sentences. However, as long as the scope of our system is not extended to the task of generating larger coherent texts, we implement a provisional solution, which has no need to look beyond sentence boundaries.

\[ \text{25 The middle affixes for subject, indirect object and direct object are not relevant here.} \]
We follow two different strategies, as the two operations of question formation versus topicalization differ in one important respect: topicalization must have a [+Foc]-element to attract, otherwise the generation process fails, whereas question formation also applies in situations without a [+WH]-element, in which case a yes/no-question is formed.

Since the [+WH]-element is not obligatory, we leave it to the user (or to chance, in case of random generation) to decide whether or not to insert such an element in an NP position in any tree.  

\[
\text{wh:: } \quad \text{WH; .}
\]
\[
\text{compl<NP>0:: } \quad \text{NP<wh>}. \]

As far as [+Foc]-elements are concerned, we do connect the generation of these to the presence of the operator Foc. We introduce a new affix with S. Again, we split up the hyperrule for S<tense> into a rule which introduces the affix terminal FOC if the attraction operator Foc is chosen, plus a rule which merely passes down this affix to a lower level. The hyperrules for S<MODAL> and S<PERF> also need to be adjusted to pass down the affix:

\[
\text{foc:: } \quad \text{FOC; .}
\]
\[
\text{S<tense,perf,foc>: } \quad ..., \text{S<type,perf,foc>, ...}
\]
\[
\text{S<ATTR>,>: } \quad \text{V<ATTR,,>,QUE>, S<type,,>;}
\]
\[
\text{V<ATTR,,>,FOC>, S<type,,FOC>}. \]
\[
\text{S<MODAL,,foc>: } \quad ..., \text{S<type,perf,foc>}. \]
\[
\text{S<PERF,,foc>: } \quad ..., \text{S<type,,foc>}. \]
\[
\text{S<PERF,NOPERF,foc>: } \quad ..., \text{S<type,,foc>}. \]

Next, we need an extra hyperrule for S<LEX> to pass down the affix FOC to one of the lexical arguments. In this rule we make use of the distribution operator (*) that GRAMGEN offers for distributing an affix over one of several sister nodes (see also the appendix). Note that this grammar only allows NPs to be [+Foc]-elements; of course, other elements can be topicalized as well, but we leave this out of account for now.

\[
\text{S<LEX,,FOC>: } \quad \text{V<LEX, su,,>, compl<su,FOC>0;}
\]
\[
\text{V<LEX, su,do,,>, compl<su,*FOC>0, compl<do,*FOC>0;}
\]
\[
\text{V<LEX, su,io,do,,>, compl<su,*FOC>0, compl<io,*FOC>0,}
\]
\[
\text{compl<do,*FOC>0.}
\]
\[
\text{compl<NP,FOC>0:: } \quad \text{NP<FOC>}. \]

---

26 Inserting [+WH]-elements independently of the presence of the operator Que has the drawback that these elements will also show up in sentences without a question operator. This will lead to the generation of sentences like the cat ate what?, which are – though not ungrammatical – contextually marked in the sense that they can only be used as echo questions.
3.2.7 Progressive

The progressive form only occurs in English (see Seuren 1996, pp.134-137). Since it appears in all four tenses (25), the progressive predicate must come below the two tense operators. This is illustrated in Figure 3.9. The predicate has as its only possible filler the auxiliary verb be, which will later trigger a transformational rule to turn the lower verb into a present participle (see section 4.3).

![Figure 3.9 The English progressive.](image)

(25) a John is working.
    b John was working.
    c John has been working.
    d John had been working.

If a level 3 adverbial is present, it follows the progressive predicate. The relevant formation rules are:

\[
S' \to V_{t2} + S^0_{PROGR} / S^0_{ADV3} / S^0 \\
S^0_{PROGR} \to V_{Progr} + S^0_{ADV3} / S^0
\]

The integration of the progressive predicate in the metagrammar is not very complicated.\(^27\)

---

\(^27\) In this implementation, the progressive can occur below a modal. In the formalization, this is indeed allowed with "regular" modals as in *John may be sleeping but not with middle auxiliaries (which we also analysed as modals, see section 3.2.4) as in *John is to be leaving (Seuren 1996, p.136).
English:

```
type::    ... [PERF] (ADV2+) << PERF << (PROGR) << [LEX] (ADV3+) << ...
tense::   ...; PROGR.
```

The position of PROGR in between the entry points [PERF] and [LEX] ensures that the progressive does not occur in tenseless complement clauses (26a), whereas it does occur in complement clauses with one (26b) or two tenses (26c).

(26)  
  a * I helped John (to) be sleeping.  
  b    I expect John to have been sleeping.  
  c    I knew that John was sleeping.

3.2.8 Passive and external dative

Two seemingly unrelated constructions are treated on a par by Seuren (1996): the passive voice, with optional by-phrase, and the external dative. The most important aspect of his treatment is that both constructions are not derived from their more neutral counterparts (the active voice and the internal dative, respectively) by means of a transformation, but that they are generated as different SAs. Thus, the two sentences in (27) each have a different underlying SA, as do the sentences in (28).

(27)  
  a Tom drove the car.  
  b The car was driven by Tom.

(28)  
  a Tom sold Harry the car.  
  b Tom sold the car to Harry.

For passives, it was already in the seventies that it became clear that they should not be derived from the active form by a transformational rule (see Bresnan 1978). Not just because such a transformation would make for a very complex rule specification, but also – more importantly – because passivization is not automatically possible with any verb that takes two or more arguments. It needs to be stated in the lexicon which verbs are allowed in a passive structure and which are not. Seuren (1985, p.188) calls it “something of an irony that this rule, once taken to be prototypical of transformational grammar, is now thought by many not to be a rule of syntax at all”. Additionally, Seuren (1998, p.234) tells us that, besides being considered a prototypical transformational rule, passive was the first transformation ever proposed (by Harris 1952).

From the Semantic Syntax point of view, the meaning differences between active and passive sentences also constitute a strong argument in favour of different underlying SAs. In general, “the Agent Phrase places greater importance on the referent of the prepositional object than a regular subject term does” (Seuren 1996, p.129); the sentences in (29) exemplify how passivization can narrow down the possible interpretation.
(29)  a  John broke a leg.
  b  A leg was broken (by John).

Seuren makes similar observations about the external and internal dative. Already in
the seventies, again, it was established that, at least in English, the two kinds of da-
tive are not at all freely interchangeable (see Green 1974). Many pairs can be shown
to differ semantically, and again “the external dative seems to bestow more impor-
tance and status on the referent of the prepositional object than the internal dative”
(Seuren 1996, p.128), which is illustrated in (30).

(30)  a  John offered the king his services.
  b  John offered his services to the king.

The relevant formation rules for English are:

\[
\begin{align*}
S' & \rightarrow V_{t1} + S^0_{PROGR} / S^0_{ADV3} / S^0_{PASS} / S^0_{DEX} / S^0 \\
S^0_{PROGR} & \rightarrow V_{progr} + S^0_{ADV3} / S^0_{PASS} / S^0_{DEX} / S^0 \\
S^0_{ADV3} & \rightarrow V_{Adv3} + S^0_{PASS} / S^0_{DEX} / S^0 \\
S^0_{PASS} & \rightarrow V_{Pass} + S^0_{PASSprep} / S^0_{DEX} / S^0 + NP \\
S^0_{PASSprep} & \rightarrow V_{Pass} + S^0_{PASS} / S^0_{DEX} / S^0 + NP \\
S^0_{DEX} & \rightarrow V_{Dex} + S^0_{DEX} / S^0_{PASS} / S^0_{ADV3} / S^0 + NP
\end{align*}
\]

These rules express the fact that the passive auxiliary (\(V_{pass}\)) comes below the second
tense operator, and also below the progressive and level 3 adverbials if these are
present. If the passive is chosen, a preposition phrase (by, \(V_{preppass}\)) is optional. The
external dative (with preposition to, \(V_{prepex}\)) comes below the passive. The SA
structures are given in Figure 3.10 and Figure 3.11.

![Figure 3.10 The passive voice.](image)

![Figure 3.11 The external dative.](image)

This can be integrated into the implemented hierarchy as follows.
We run into two problems now, the first of which relates to the entry point [LEX]. The fact that PASS, PASSP and DEX are placed below ADV3 in the hierarchy implies that they come below the entry point [LEX] as well (for ADV3 comes below [LEX]; see section 3.2.5). This means that both passive and the external dative should be allowed in a tenseless complement clause. Whereas this is indeed the case for the external dative (31a), it is not true for passive (31b). Seuren’s formalization does not solve this paradox. Since a level 3 adverbial can occur in a tenseless complement clause, he must say that “a complement $S^0$ may enter rule (3c)” (p.89), where rule (3c) refers to the rule rewriting $S^{0}_{ADV3}$ as formulated above. But at the same time he must stipulate that “genuine $S^0$-embeddings (...) cannot take the superscripts for (...) the passive voice” (p.90).

(31) a  I saw Tom sell the car to Harry.
      b  * I saw the car be driven (by Tom).

Obviously, this leads us to consider whether PASS and PASSP can instead be situated above ADV3, and thus also above [LEX]. As we will see below, this is actually the case in Dutch and German – so why not in English? This has to do with the relative order of the preposition phrases involved. Example (32) shows that, when combining external dative, passive and level 3 adverbial, the correct surface structure order is: to-phrase, by-phrase, manner preposition phrase (or manner adverb, for that matter).\textsuperscript{28} This surface order results from a transformational operation on the elements involved, called Lowering (see also section 4.3.2). Without going into details now, it suffices to say that the effect of this operation is that the elements end up in the reverse order. Hence, the underlying order in the SA should be: level 3 adverbial, passive preposition, external dative.

(32) The letter was written to Harry by Tom with accuracy.

For now, this leaves us with the choice between an unwanted passive in tenseless complement clauses, and the wrong order when combining a by-phrase with a level 3 adverbial. We decide on the first option, since this is also what is chosen in Seuren’s formalization. In other words, at this stage we do not solve the first problem of the hierarchy proposed – rather, it remains a matter for further research.

The second problem of our implementation is of a purely technical nature and has to do with the nested optionality brackets. Recall that we used nested brackets before, namely with level 2 adverbials to indicate that these can only be chosen if a modal is chosen too (see section 3.2.5):

\textsuperscript{28} Note that Seuren assumes the to- and by-phrases to be in reverse order on p.101, whereas on p.136 he assumes the order in (32). Perhaps this indicates that the order is not as fixed as we might think?
The same sort of dependent optionality occurs with the passive preposition, which can only be chosen if a passive auxiliary is chosen. The difference is that, whereas the optional ADV2 comes above MODAL, the optional PASSP comes below PASS:

\((\text{PASS} \ll (\text{PASSP}))\)

This difference is precisely the problem. For technical reasons, GRAMGEN cannot interpret nested optionality brackets correctly if they are the final element within the higher brackets. More generally, optionality brackets may never end a hierarchy or a subhierarchy (note that the final element in our complete hierarchy always is the obligatory LEX).

Although this technical problem seems to call for a technical solution, it also brings up the theoretical question whether the order should necessarily be auxiliary-preposition. Could it be preposition-auxiliary instead? Semantically, we do not see a reason to prefer either of the two orders in the SA: there seem to be no meaningful scope differences involved. Syntactically, it also has no consequences if we change the relative order of the two elements. As mentioned above, the preposition induces a transformation called Lowering. This results in the by-phrase being moved out of the way completely, so that it has no way of interfering with the transformations operating on the auxiliary.

Therefore, we can solve our technical problem by means of an adjustment to the formalization which is otherwise theoretically neutral. We change the relative order of passive auxiliary and passive preposition.

**English:**

\[ [\text{LEX}] (\text{ADV3}^*) \ll ( (\text{PASSP}) \ll \text{PASS}) \ll (\text{DEX}) \ll \text{LEX}. \]

Now let us turn to Dutch and German. The formation rules are:

**Dutch:**

\[
\begin{align*}
S' & \rightarrow V_{12} + S^{0}\text{PASS} / S^{0}\text{ADV3} / S^{0}\text{DEX} / S^0 \\
S^{0}\text{PASS} & \rightarrow V_{\text{Pass}} + S^{0}\text{PASSPrep} / S^{0}\text{ADV3} / S^{0}\text{DEX} / S^0 \\
S^{0}\text{PASSPrep} & \rightarrow V_{\text{PrepPass}} + S^{0}\text{ADV3} / S^{0}\text{DEX} / S^0 \\
S^{0}\text{ADV3} & \rightarrow V_{\text{Adv3}} + S^{0}\text{ADV3} / S^{0}\text{DEX} / S^0 \\
S^{0}\text{DEX} & \rightarrow V_{\text{PrepDev}} + S^{0} / S^{0} + \text{NP}
\end{align*}
\]

**German:**

\[
\begin{align*}
S' & \rightarrow V_{12} + S^{0}\text{PASS} / S^{0}\text{ADV3} / S^0 \\
S^{0}\text{PASS} & \rightarrow V_{\text{Pass}} + S^{0}\text{PASSPrep} / S^{0}\text{ADV3} / S^0 \\
S^{0}\text{PASSPrep} & \rightarrow V_{\text{PrepPass}} + S^{0}\text{ADV3} / S^0 + \text{NP}
\end{align*}
\]
The differences between the three languages amount to the following. As we explained above, in English both the passive and the external dative come below the level 3 adverbials, whereas in Dutch and German the passive precedes these adverbials. Furthermore, the German language has no external dative. The integration into the implemented hierarchies is straightforward, again adjusting the order of the passive auxiliary and passive preposition:

**Dutch:**

```plaintext
type:: ...
[PERF] (ADV2*) << PERF << ((PASSP) << PASS) <<
([LEX]) (ADV3*) << (DEX) << LEX.
```

**German:**

```plaintext
type:: ...
[PERF] (ADV2*) << PERF << ((PASSP) << PASS) <<
([LEX]) (ADV3*) << LEX.
```

Note that the problem of an unwanted passive in tenseless complement clauses does not occur in Dutch and German, because PASSP and PASS can simply precede the entry point [LEX] since they also precede ADV3.

The theoretical parallels between the passive versus active voice and the external versus internal dative, regarding the selection of lexical arguments, are reflected in the formalization. If a passive or external dative is chosen, this will have an effect further down in the tree. The effect is brought about by means of the subscript feature [PASS] or [DEX], respectively. Seuren explains how these features work:

> "A subscript feature [a] is passed on to subsequent Ss until bare Sⁿ[a] is reached. Then the feature [a] is passed on to the lexical verb selected, where it first ensures that a lexical verb is selected from the appropriate class of verbs and then makes for an appropriate gap in the lexical argument frame selection. That is, if the feature is [PASS] a verb is to be selected from the class of verbs that allow for passivization, and the argument frame selection is made without the subject term. If the feature is [DEX] a verb is to be selected from the class of verbs that allow for an indirect object, and the lexical argument frame selection is made without the indirect object term. Then, the object-NP of passive by is assigned the feature [SUₜₚ], i.e. selection is to be made as if it were the subject of the lexical verb, and the object-NP of dative to inherits the feature [IOₜₚ], i.e. selection is to be made as if it were the indirect object of the lexical verb. It is important to note that features that have been created by the Formation Rules do not affect the application of the rules: the Formation Rules are not sensitive to features. (...) Features acquired in the course of the generation process accumulate." (Seuren 1996, p.100)

In other words, the effect of passive and external dative on the argument selection is not spelled out in the formation rules, but formalized as a separate mechanism superimposed on these rules.

The features [PASS] and [DEX] can be reflected quite easily in the implementation. We use the selectional affix we introduced earlier in the context of middle auxiliaries for passing down restrictive information to a lower level (see section 3.2.4). The selectional affix terminals PASS and DEX are introduced on the sister-S of the passive auxiliary and the dative preposition, respectively. This is cumulative, hence the concatenation operator and the affix terminal PASSDEX (for the technical details of
the operator, see the appendix). The regular rewrite rule for $S<\text{tense}>$ merely passes down the information through the affix terminal select:

\[
\text{tense}:: \quad \ldots; \text{PASSP}.
\]

\[
S<\text{tense}, \text{select}, \text{foc}>: \ldots, S<\text{type}, \text{select}, \text{foc}>, \ldots
\]

**English and Dutch:**

\[
P\text{ass}/d\text{ex}:: \quad \text{PASS}; \text{DEX}.
\]

\[
\text{select}:: \quad \text{NOPERF}; \text{PASS}; \text{DEX}; \text{PASSDEX}; .
\]

\[
S<P\text{ass}/d\text{ex}, \text{select}, \text{foc}>:
\]

\[
V<P\text{ass}/d\text{ex}, , , \text{optdo}, >, S<\text{type}, \text{select+pass}/d\text{ex}, \text{foc}>, \text{compl}<\text{optdo}, >0.
\]

**German:**

\[
\text{select}:: \quad \text{NOPERF}; \text{PASS}; .
\]

\[
S<\text{PASS}, \text{select}, \text{foc}>:
\]

\[
V<\text{PASS}, , , \text{optdo}, >, S<\text{type}, \text{select+PASS}, \text{foc}>, \text{compl}<\text{optdo}, >0.
\]

Finally, we expand the rewrite rules for $S<\text{LEX}>$ for all four possible situations (it is expressed in the lexicon which predicates allow for passivization and/or external datives). Without PASS or DEX any lexical predicate can be chosen, with its full argument frame:

\[
P\text{ass}:: \quad \text{PASS}; .
\]

**English and Dutch:**

\[
d\text{ex}:: \quad \text{DEX}; .
\]

\[
S<\text{LEX}, , , >: \quad V<\text{LEX}, \text{su}, \text{optio}, \text{optdo}, \text{pass+dex}>, \ldots
\]

\[
S<\text{LEX}, , , \text{FOC}>: \quad V<\text{LEX}, \text{su}, , , \text{pass+dex}>, \ldots;
\]

\[
V<\text{LEX}, \text{su}, , , \text{do}, \text{pass+dex}>, \ldots;
\]

\[
V<\text{LEX}, \text{su}, \text{io}, \text{do}, \text{pass+dex}>, \ldots
\]

**German:**

\[
S<\text{LEX}, , , >: \quad V<\text{LEX}, \text{su}, \text{optio}, \text{optdo}, \text{pass}>, \ldots
\]

\[
S<\text{LEX}, , , \text{FOC}>: \quad V<\text{LEX}, \text{su}, , , \text{pass}>, \ldots;
\]

\[
V<\text{LEX}, \text{su}, , , \text{do}, \text{pass}>, \ldots;
\]

\[
V<\text{LEX}, \text{su}, \text{io}, \text{do}, \text{pass}>, \ldots
\]

With PASS only lexical predicates that allow for passivization can be chosen and the subject is left out.

**English and Dutch:**

\[
S<\text{LEX}, \text{PASS}, , >:
\]

\[
V<\text{LEX}, \text{su}, \text{optio}, \text{do}, \text{PASS+dex}>, \text{compl}<\text{optio}, >0, \text{compl}<\text{do}, >0.
\]

\[
S<\text{LEX}, \text{PASS}, \text{FOC}>:
\]

\[
V<\text{LEX}, \text{su}, , , \text{do}, \text{PASS+dex}>, \text{compl}<\text{do}, \text{FOC}>, 0;
\]

\[
V<\text{LEX}, \text{su}, \text{io}, \text{do}, \text{PASS+dex}>, \text{compl}<\text{io}, \text{FOC}>, 0, \text{compl}<\text{do}, \text{FOC}>, 0.
\]
German:
\[ S<\text{LEX}, \text{PASS}, >: \]
\[ V<\text{LEX}, \text{su}, \text{optio}, \text{do}, \text{PASS}>, \text{compl}<\text{optio}, >0, \text{compl}<\text{do}, >0. \]
\[ S<\text{LEX}, \text{PASS}, \text{FOC}>: \]
\[ V<\text{LEX}, \text{su}, , \text{do}, \text{PASS}>, \text{compl}<\text{do}, \text{FOC}>0; \]
\[ V<\text{LEX}, \text{su}, \text{io}, \text{do}, \text{PASS}>, \text{compl}<\text{io}, \text{FOC} >0, \text{compl}<\text{do}, \text{*FOC} >0. \]

With DEX only lexical predicates that allow for an external dative can be chosen and the indirect object is left out. With PASSDEX only lexical predicates that allow for passivization and for an external dative can be chosen and both the subject and the indirect object are left out.  

English and Dutch:
\[ S<\text{LEX}, \text{DEX}, >: \]
\[ V<\text{LEX}, \text{su}, \text{io}, \text{do}, \text{pass+DEX}>, \text{compl}<\text{su}, >0, \text{compl}<\text{do}, >0. \]
\[ S<\text{LEX}, \text{DEX}, \text{FOC}>: \]
\[ V<\text{LEX}, \text{su}, \text{io}, \text{do}, \text{pass+DEX}>, \text{compl}<\text{su}, \text{*FOC} >0, \text{compl}<\text{do}, \text{*FOC} >0. \]
\[ S<\text{LEX}, \text{PASSEDEX}, \text{foc}>: \]
\[ V<\text{LEX}, \text{su}, \text{io}, \text{do}, \text{PASSEDEX}>, \text{compl}<\text{do}, \text{foc} >0. \]

3.2.9 Noun phrases

Seuren does not give a very detailed analysis of the internal syntax of NPs. Figure 3.12.a shows the basic structure of NPs in Semantic Syntax (see also Seuren 1996, p.101). The formation rules defining this structure are:

\[ \text{NP} \rightarrow :x^/\sim x+S^{\text{NOM}} \]
\[ \text{or: } x \]
\[ \text{or: } X_{\text{WH}} \]
\[ S^{\text{NOM}} \rightarrow V_{\text{Nom}}+\text{NP}[x] \]
\[ \text{or: } V_{\text{Adj}}+S^{\text{NOM}} \]

The internal structure of NPs is motivated primarily from a model-theoretical (i.e. semantic) point of view. The \( S^{\text{NOM}} \)-subtree denotes the set of all items for which the nominal predicate and, if present, the adjectival predicates hold. So, in Figure 3.12.a, the lowest \( S^{\text{NOM}} \) denotes the set of all men, and the higher \( S^{\text{NOM}} \) denotes the set of all old men.  

The elements \( :x \) and \( \sim x \) are model-theoretical operators. The colon operator binds \( x \) and selects one individual from the \( S^{\text{NOM}} \)-set, leading to a definite NP ("the old man"). The cap operator is semantically vacuous, in the sense that it only turns the S-structure into an NP-structure, while the denotation remains

---

29 Note that we do not incorporate Seuren’s features \([S\times_{\text{LEX}}] \) and \([I\times_{\text{LEX}}] \) into the implementation. Although it would be quite easy to do so we have decided not to, because the selection criteria that are supposed to be triggered by these features are not implemented (or formalized) either.

30 More precisely, the higher \( S^{\text{NOM}} \) should be understood as something like the set of "all x’s that man oldly". See Seuren (1996, p. 310-311) for the arguments behind this unorthodox analysis of attributive adjectives.
the same (combined with a quantifier this leads to an indefinite NP, e.g. "an old man"; see also the next section).

To keep the internal NP-structure more in line with the rest of the grammar, we make an adjustment to it as illustrated in Figure 3.12.b. The operator :x//x must have a category label, since it is a filler and all fillers have category labels (compare our discussion of the embedded question operator on p.47). We opt for the label \( V_x \), since all other operators in the grammar are of type \( V \) as well.

![Diagram](image)

Figure 3.12 The skeletal structure of NPs.

Now, the formation rules can be matched onto GRAMGEN rules quite closely:

\[
\begin{align*}
& \text{var::} & \text{VAR;} & . \\
& \text{NP<foc>::} & \text{V<x,/>,,DEF>, S<NOM,,/>} & . \\
& \text{NP<var>::} & "x". \\
& \text{NP<WH>::} & \text{V<PRON,,/>,,WH>}. \\
& \text{S<NOM,,/>::} & \text{V<LEX,NOM,,/>,,>, S<NOM,,/>;} \\
& & \text{V<NOM,,/>,,>, NP<VAR>}. \\
\end{align*}
\]

Since the indefinite operator ^x (represented in the lexicon as \( V<x, \text{INDEF}> \)) only appears in the scope of a quantifier (see the next section), we restrict generation within a normal, non-quantified NP to the definite operator :x (represented as \( V<x, \text{DEF}> \)). The affix nonterminal foc determines whether the NP is to be attracted by the topicalization operator (see section 3.2.6). The operator S\(^{\text{NOM}}\) is represented in the lexicon as \( S<\text{NOM}> \), \( V_{\text{Adj}} \) as \( V<\text{LEX, NOM}> \)\(^{31}\) and \( V_{\text{Nom}} \) as \( V<\text{NOM}> \). We use the affix terminals \( \text{VAR} \) and \( \text{WH} \) to identify the variable filler \( x \) and the wh-pronouns, which we list in the lexicon under \( V<\text{PRON, WH}> \).

\(^{31}\) \( V_{\text{Adj}} \) is not simply represented as \( V<\text{ADJ}> \) because adjectives can also function as lexical predicates in the nucleus, i.e. they cannot only be used attributively but also predicatively. As lexical predicates they need to be represented as \( V<\text{LEX}> \).
In the absence of any principled treatment by Seuren of bare nominal elements like pronouns and proper names, we use our own provisional strategy for incorporating these elements into the implementation. On a par with the wh-pronouns, we list them in the lexicon as \(\text{V}<\text{PRON}>\) and extend the rewrite rule for \(\text{NP}<\text{foc}>\):

\[
\text{NP}<\text{foc}>: \quad \text{V}<\text{X}, \ldots, \text{DEF}>, \text{S}<\text{NOM}, >; \\
\text{V}<\text{PRON}, \ldots, >.
\]

An additional piece of information that needs to be supplied with some NPs, is a coreference index. Some lexical predicates with S-objects induce cyclic deletion of the subject of the embedded clause (for further details, see section 4.3.7; see also Seuren 1996, p.68-71). This deletion is bound by a controller: an NP-argument of the lexical predicate, which must have anaphoric referential identity with the embedded subject that is to be deleted.

"The referential identity is expressed in tree structures by subscripting the controller’s NP-label with a variable symbol (x, y, or z) and providing the same variable symbol as filler for the lower subject NP that is up for deletion."

(Seuren 1996, p.68)

In the implementation, this variable subscript is translated into an NP affix terminal \(X\), which shares the affix position with \(\text{FOC}\) (both affixes only determine how the NP is to be treated transformationally and have no effect on the internal structure of the NP).\(^\text{32}\)

\[
\text{nptype}: \quad X; \text{FOC}; .
\]

\[
\text{compl}<\text{NPX}, >0: \quad \text{NP}<\text{X}>.
\]

\[
\text{NP}<\text{nptype}>: \quad \text{V}<\text{X}, \ldots, \text{DEF}>, \text{S}<\text{NOM}, >; \\
\text{V}<\text{PRON}, \ldots, >.
\]

A separate type of NPs is illustrated in Figure 3.13. Seuren makes a distinction between bare complement-Ss and complement-Ss headed by an NP-node. Syntactically, the main difference is that the latter can be fronted, whereas the former cannot. This is illustrated in (33) and (34): \textit{seem} has an S'-subject which cannot be fronted. and \textit{follow} has an \(\text{NP}[\text{S}']\)-subject which can be fronted. An \(\text{NP}[\text{S}]\) has the internal structure of an S, but – in some respects – the distribution of an NP. Semantically, the NP-status of a complement clause often corresponds to factivity.

(33) a It seems that Tom is ill.
    b * That Tom is ill seems.

(34) a It follows that Tom is ill.
    b That Tom is ill follows.

Seuren only has examples of \(\text{NP}[\text{S}']\)-arguments in the lexicons – no \(\text{NP}[\text{S}]\) or \(\text{NP}[\text{S}^0]\), although these also occur in the languages under investigation. The implementation of the \(\text{NP}[\text{S}']\)-complements is straightforward:

\(^{32}\) Since \textsc{gramgen} provides no simple means for keeping track – during the generation process – of what variables are already in use, we assume for the time being that we will not need more than one variable per tree.
3.2.10 Quantifiers

Seuren (1996, p.300-309) treats quantified NPs differently from the non-quantified ones, which receive the analysis as outlined in the previous section. The main semantic difference between the two is that quantified expressions are sensitive to scope (35), whereas non-quantified expressions are not (36). Seuren expresses this scope sensitivity in the SA-structure by factorizing the quantifiers out. In (35), the a and b examples are ambiguous to most native speakers of English, but in both cases the preferred reading is clearly the one in which the first quantifier has scope over the second quantifier. So (35a) predominantly means something like “for every student it holds that he/she speaks two languages”, whereas (35b) first and foremost means “there are two languages such that every students speaks them”. Obviously, this ambiguity and scope sensitivity does not occur in (36a) and (36b).

(35)  a  All students speak two languages.
      b  Two languages are spoken by all students.

(36)  a  This girl speaks two languages.
      b  Two languages are spoken by this girl.

(37)  a  My parents speak two languages.
      b  Two languages are spoken by my parents.

Note that a problem with respect to scope sensitivity arises if we look at regular plural NPs, which are normally not treated as quantified expressions but do appear to be sensitive to scope. The sentences in (31) reveal the same ambiguity and scope sensitivity pattern as those in (29). This may pose a problem for Semantic Syntax, as Seuren provides no means to account for these facts – he does not account for plural NPs at all. The solution would most likely be directed towards splitting up plural nouns into a lexical part and a scopal operator as well. We will, however, not investigate this matter any further, leaving it open for further study.
The places a quantifier can occupy in the SA are the same as the places of level 2 adverbials in English, and the level 1 and 2 adverbials in Dutch and German:

“Since the class of \( V_{abh2} \) contains logical scope-bearing adverbs like not, quantifiers can occur, in SA-structures, in exactly the positions that the negation and other logical scope-bearing adverbs can occur in, thus varying their scope relations with respect to each other.”

(Seuren 1996, p.99)

The relevance of scope relations between quantifiers and level 2 adverbials is illustrated in (38).

(38)  
a. All my friends will not forget my birthday.  
b. Not all my friends will forget my birthday.

An SA-structure with a quantifier looks as in Figure 3.14 (with Figure 3.14.a depicting the general scheme and Figure 3.14.b depicting the SA of the sentence All students speak French). The quantifier selects an NP-object with an obligatory cap operator ("\( x \), see also the previous section). The S-subject, which follows the normal S-type hierarchy, is headed by an NP-node and has the same cap operator as its left sister. One of the NP-arguments of the lexical predicate has a variable filler \( x \).

\[
\text{Figure 3.14 A quantified SA.}
\]

Semantically, this is to be interpreted as the quantifier being a binary predicate over pairs of sets, i.e. a higher order predicate. For example, the meaning of the existential quantifier \( a \) is: the intersection of the subject term denotation and the object term

---

33 Note that in Figure 3.14, the structure of the NP is as in Figure 3.13.b, i.e. with a category label for the cap operator.
denotation is non-empty. This analysis of quantifiers is known from the theory of
generalized quantification (Mostovski 1957, Barwise & Cooper 1981). Note, how-
ever, that in Seuren’s SAs, as opposed to most other treatments of generalized
quantifiers, the quantified NP is the rightmost argument, with the matrix NP between
it and the quantifier (see also Seuren 1996, p.302-303).

Seuren’s formation rules defining this structure are:

<table>
<thead>
<tr>
<th>English</th>
<th>Dutch and German</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S^n_{ADV2} \rightarrow V_{Adv2} + S^n_{ADV2} / S''$</td>
<td>$S^n_{ADV1} \rightarrow V_{Adv1} + S^n_{ADV1} / S''$</td>
</tr>
<tr>
<td>or: $V_{Prep3} + S^n_{ADV2} / S'' + NP$</td>
<td>or: $V_{Prep1} + S^n_{ADV1} / S'' + NP$</td>
</tr>
<tr>
<td>or: $V_0 + NP[\langle x+S^n_{ADV2} / S'' \rangle + NP$</td>
<td>or: $V_0 + NP[\langle x+S^n_{ADV1} / S'' \rangle + NP$</td>
</tr>
<tr>
<td>or: $V_{Prep2} + S^0_{ADV2} / S' + NP$</td>
<td>or: $V_{Prep2} + S^0_{ADV2} / S' + NP$</td>
</tr>
<tr>
<td>or: $V_0 + NP[\langle x+S^0_{ADV2} / S' \rangle + NP$</td>
<td>or: $V_0 + NP[\langle x+S^0_{ADV2} / S' \rangle + NP$</td>
</tr>
</tbody>
</table>

However, these rules lack two restrictions on quantified SAs. First, they do not ex-
press the fact that the object NP of the quantifier must have a cap operator $^\ast x$. And
second, they do not force one of the NP-arguments of the lexical predicate to have
the filler $x$, which is the variable to be bound by the quantifier.

By choosing the following implementation, we easily eliminate the first shortcoming
from the formalization. Quantifiers are represented in the lexicon as $V<ADV1, NP, NP>$ or $V<ADV2, NP, NP>$ (an adverbial of level 1 or 2, but
unlike the adverbs and prepositions of that level with an NP-subject and an NP-
object).

$S<\text{quant, select, foc} > :$

$V<\text{quant, NP, NP, }>, \ \text{NP}<\langle x+\text{type+select+foc} \rangle , \ \text{NP}<^\ast x >.$

**English:**

| quant:: | ADV2. |

**Dutch and German:**

| quant:: | ADV1; ADV2. |

Both NPs have the cap operator $^\ast x (\langle ^\ast x \rangle )$, represented in the lexicon as $V<x, \text{INDEF}>$. Additionally, the subject NP is to continue the S-type hierarchy, so
it passes on the affixes type, select and foc to its daughter S.

$\text{NP}<^\ast x > : \ \ \ V<x, , , \text{INDEF}>, \ S<\text{NOM, }>, .\$

$\text{NP}<\langle x+\text{type+select+foc} \rangle > : \ V<x, , , \text{INDEF}>, \ S<\text{type, select, foc} >.$

The second shortcoming is reminiscent of the question we encountered in the con-
text of the attraction operator, which was how to enforce the generation of the elements that were to be attracted (see section 3.2.6). We follow the same strategy
here as with the focus elements above: the generation of the lower element (the vari-
able $x$ and the $[+\text{Foc}]$-element, respectively) must depend upon the presence of the higher element (the quantifier and the attraction operator, respectively). Within the
subject NP of a quantifier, we introduce the affix <VAR> on the S-node, at the same position as <FOC>. This affix is passed on by all rules expanding an S.

NP<‘X+type+select+foc+VAR>: ..., S<type, select, foc+VAR>.
S<tense, select, foc+VAR>: ..., S<type, select, foc+VAR>, ...
S<MODAL, , foc+VAR>: ..., S<type, perf, foc+VAR>.
S<PERF, , foc+VAR>: ..., S<type, , foc+VAR>.
S<PERF, NOPERF, foc+VAR>: ..., S<type, , foc+VAR>.

**English and Dutch:**
S<pass/dex, select, foc+VAR>: ...
   ..., S<type, select+pass/dex, foc+VAR>.

**German:**
S<PASS, select, foc+VAR>: ..., S<type, select+PASS, foc+VAR>.

Also, we need to introduce extra nucleus rules again, but thanks to GRAMGEN’s distribution operator (see the appendix) we can keep the increase of the number of rules within limits:

foc/Var:: FOC; VAR.

S<LEX, , foc/Var>:
   ..., complsu, foc/Var>0;
   ...
   ...
   ...
   complsu, 'foc/Var>0, complndo, 'foc/Var>0;
   ...
   complsu, 'foc/Var>0, complnio, 'foc/Var>0,
   complndo, 'foc/Var>0.
S<LEX, PASS, foc/Var>:
   ...
   ...
   ...
   complndo, foc/Var>0;
   ...
   complnio, 'foc/Var>0, complndo, 'foc/Var>0.

**English and Dutch:**
S<LEX, , FOCAVAR>:
   V<LEX, su, , pass+dex>, complsu, FOCAVAR>0;
   V<LEX, su, do, pass+dex>, complsu, *FOC*+VAR>0,
   complndo, *FOC*+VAR>0;
   V<LEX, su, io, do, pass+dex>, complsu, *FOC*+VAR>0,
   complnio, *FOC*+VAR>0, complndo, *FOC*+VAR>0.
S<LEX, PASS, FOCAVAR>:
   V<LEX, su, do, PASS+dex>, complndo, FOCAVAR>0;
   V<LEX, su, io, do, PASS+dex>, complnio, *FOC*+VAR>0,
   complndo, *FOC*+VAR>0.
S<LEX, DEX, foc/Var>:
   ...
   ...
   ...
   complsu, 'foc/Var>0, complndo, 'foc/Var>0.
S<LEX, DEX, FOCAVAR>:
   V<LEX, su, io, do, pass+DEX>, complsu, *FOC*+VAR>0,
   complndo, *FOC*+VAR>0.
S<LEX, PASSDEX, foc+VAR>:
   ..., complndo, foc+VAR>0.
3.3 The lexicon

Figure 3.15, another excerpt from Figure 2.1, depicts how the implemented lexicon is interconnected to the other modules of our generator. There is one lexicon file per language. This root file is compiled into two separate lexicons, based on two general frames. One of these lexicons contains the information that is needed by the formation rules; the other one holds the information needed by the transformational grammars. The present section describes the root lexicon file, which I will simply refer to as the lexicon from now on.

The implemented lexicon closely resembles that of Seuren (1996). It consists of a number of columns, listing various attributes of the predicates.

The first column states what type of predicate we are dealing with; Seuren calls this the SA-category of the predicate. However, as we have argued in the context of prepositions and adverbs (see section 3.2.5), the concept of grammatical category only becomes relevant at the surface level. To avoid any confusion, we therefore prefer to use another heading for this column: Class. The entries in this column are the <type>/<tense> affixes used in the formation rules, for example ADV0, FIN, LEX.

The second column lists the Fillers, including not only the actual words that make up the sentence (moreover, seem, help et cetera), but also abstract operators such as QUE, PRES, PAST.
The third, fourth and fifth columns list the predicate’s lexical argument frame: its subject (Su), indirect object (Io), and direct object (Do), respectively. Seuren lists the three arguments together in one column, but we split the frame up as this makes the integration with the formation rules – in which three separate V affixes are reserved for the arguments – easier. Another difference with the formalization is that Seuren states the argument frame only for the lexical predicates (nouns, adjectives and verbs), whereas we also list the arguments for higher predicates such as the adverbials and tense operators. We have two reasons for doing so. One is simply that we prefer not to split up the lexicon into two separate parts without having any other, independent grounds for doing so. The other reason is that we can in fact make use of the subject and direct object slots for some of the higher predicates. We use the direct object slot to distinguish the prepositions from the adverbs (see section 3.2.5), and the subject slot to identify the quantifiers (see section 3.2.10) and the middle auxiliaries (see section 3.2.4). The entries in these columns are affix terminals when only one option is possible (for example, NP for an argument noun phrase or LEX for an argument clause without a tense operator), or affix nonterminals when the predicate can have different subjects, indirect objects or direct objects (for example, fin/perf when the argument clause has either one or two tense operators).
The sixth column states the surface category (Cat) of each predicate, just as with Seuren. Examples are ADV for adverbs, AFF for morphological affixes and V for verbs. This information is retrieved from the lexicon in the transformational grammar (see section 4.3.4).

The next column lists the cyclic rules (Cyclic rules) that are induced by each predicate, in the same way as is done in the formalization. So, for example, L<S> means that Lowering must be applied to the relevant predicate with landing site S (see section 4.3.2), SR means that Subject Raising must take place (see section 4.3.5), and SD means that Subject Deletion is to be applied (see section 4.3.7). The transformational grammar retrieves this information from the lexicon (see section 4.2.7).

Then there is a column called Restr, containing the affix terminals related to various restrictions in the formation rules: QUE and FOC to distinguish the two attraction operators (see section 3.2.6), DEF and INDEF to distinguish the two X operators (see sections 3.2.9 and 3.2.10), NOPERF to distinguish the perfective tense operator SIM from the other one (PRET, see section 3.2.4), and PASS, DEX and PASSDEX to identify the verbs that allow for passivization and/or an external dative (see section 3.2.8). This column is unique to the implementation, since in the formalization the relevant mechanisms were either left implicit or realized in a different way.

The last column (Extra) is something of a repository for lexical information needed in the transformational grammar and expressed by Seuren in some other way. This concerns exceptional lexical argument functions (the middle argument being a direct object, or the final argument an indirect object – af<DO> and af<IO>, respectively; see sections 4.2.4 and 5.5.1), expressed by Seuren as a feature within the lexical argument frame itself: the irregular applicability of the transformational rule Particle-insertion (to/-to, -te, +zu; see sections 4.3.6 and 5.5.5), expressed by Seuren as a feature of the cyclic rules; the exceptional behaviour of a few adverbs when it comes to Lowering and Fronting (NOSTOP; see sections 4.3.2 and 4.4.4), expressed by Seuren by means of a symbol † prefixed to the filler; the choice of some Dutch and German verbs of the perfective auxiliary zijn or sein rather than hebben or haben (BE; see section 5.5.2), expressed by Seuren by means of an asterisk prefixed to the filler. All this is retrieved from the lexicon by the transformational grammar.

With these nine columns, the lexicon looks as follows:\textsuperscript{34}

\textsuperscript{34} For a full specification of all three implemented lexicons, see the CD-ROM accompanying this thesis.
3.4 Conclusion

To conclude this chapter, let me summarize what we claim to have contributed to the model of Semantic Syntax by implementing its formation rules and lexicon. First, let me recall our research questions, as formulated in chapter 1. With respect to the descriptive adequacy of Semantic Syntax, we were interested to know whether the grammars as formalized by Seuren (1996) are internally consistent and account for exactly the set of sentences that they are claimed to account for. Additionally, from a technolinguistic viewpoint, we wanted to establish if Seuren’s formalization is a descriptively optimal representation of the underlying theory. With respect to the crosslinguistic adequacy of Semantic Syntax, we resolved to find theoretical generalizations about the interrelation between the grammars for English, Dutch and German.

Overall, I think it is fair to say that we have succeeded in implementing the Semantic Syntax formation rules and lexicon satisfactorily. We have managed to build two principle-based modules that remain very close to Seuren’s formalization and that generate the SA-structures of all the sentences that Semantic Syntax is claimed to cover so far. In the next two sections, I will refine this general conclusion and give a balanced synopsis of the various challenges we have met with respect to the model’s descriptive and crosslinguistic adequacy.

3.4.1 Descriptive adequacy

Inconsistencies

With respect to the descriptive adequacy, we have detected a number of problems, lacunae and inconsistencies in Seuren’s formalization. In some cases, we have been able to solve these problems by turning to the underlying theory. In other cases, when the theory did not provide an answer, we have either solved the problem ourselves, or have left it for further study. In general, we have only tried to solve a problem when it formed an obstacle for the generation of sentences that Seuren claims to account for.

A case where the formalization could be seen to deviate from the theory concerns the general SA format (see section 3.2.2). This format, as given by Seuren (1996,
p.25), states that every S-node expands into a V-node with one, two or three arguments, the first argument being the subject, the last argument being the direct object and the middle one being the indirect object. Although Seuren’s formation rules abide by the general format, this is more or less coincidental in the sense that the chosen formalization does not intrinsically force the format upon the rules. In our implementation, on the other hand, we have expressed the format more explicitly, albeit a weaker version of it. One of our reasons for settling for a weaker version is that the general format does not hold for all lexical predicates (see footnote 14, p.35).

An example of an inconsistency within the formalization that we have removed by looking at the theory is formed by the fillers of the tense operators (see section 3.2.1). In Seuren's formalization, the first tense operator has an abstract filler (PRES or PAST), but the second operator is filled by either ∅ or a perfective auxiliary verb (have, hebben/zijn or haben/sein). In the underlying theory, however, both operators contribute in a similar way to a Reichenbachian tense analysis (V₁₁ stands for the relation between time of speaking and reference time, V₁₂ for the relation between reference time and event time). Thus, we have removed the inconsistency by replacing Seuren's V₁₂-fillers with the abstract fillers SIM and PREC, parallel to the V₁₁-fillers PRES and PAST.

Another internal inconsistency that we have removed from the formation rules is Seuren's representation of dependent questions. We have replaced his rudimentary specification _NP[Que+S"]_ by the more complex but also more accurate _NP[S+Amp[Var][Que]+S"]_ (see section 3.2.6). This is a typical case where we were forced to be more precise just by the process of implementing. A similar adjustment has been made to the internal structure of regular NPs, which we have changed from _NP[/:/x+S\_NOM]_ into _NP[\[\lambda x:/:/x\]+S\_NOM]_ (see section 3.2.9).

An aspect which Seuren, as he recognizes himself, has not formalized is the origin of the elements to be attracted by the question or topicalization operator, i.e. the elements marked [+WH] or [+FOC]. On a theoretical level, Seuren assumes that “external factors, to do with discourse and situation are responsible for the feature” (1996, p.148). Since further study of these factors falls well beyond the scope of the present thesis, they are not incorporated in the implementation either. All that we have implemented is a provisional way to introduce the elements into the SA (see section 3.2.6). The same strategy was followed to ensure the generation of a variable within a quantifier’s scope (see section 3.2.10).

A similar situation (which we have solved however) occurs with the definite and indefinite operators */x and :x. The indefinite operator should only occur in the scope of a quantifier, but Seuren does not account for this in the formalization: his formation rules can generate indefinite and definite NPs in any configuration, regardless of whether a higher quantifier is present to bind the operator. In the implementation, we have restricted generation within non-quantified NPs to the definite operator :x, and
generation within NPs in the scope of a quantifier to the indefinite operator ∧x (see sections 3.2.9 and 3.2.10).

A problem in the formalization that we have not solved is that the formation rules allow for a passive in tenseless complement clauses in English. We have shown that this is not correct, as such complement clauses are ungrammatical. However, we have also shown that the only straightforward solution to this would cause another, more serious problem (i.e. the wrong order of passive by-phrases and level 3 adverbials). Hence, we have left the first problem unsolved and open for further study (see section 3.2.8).

Also not implemented ideally is the coreference index that is supposed to indicate coreferentiality between an NP and its controller in the higher clause. Due to limitations of our program tool we can only generate sentences with at most one reference variable (see footnote 32, p.60). This is enough, however, for the sentences that Semantic Syntax is claimed to account for.

An aspect that Seuren seems to have formalized concerns the choice of the arguments of passive by and external dative to. The features [SUvlec] and [IOvlec] are supposed to impose the correct restrictions on these arguments, i.e. the same restrictions as imposed on the subject or indirect object of the lexical verb in question. However, such restrictions are not worked out in the formalization at all, so in reality they can neither be transferred to the by- and to-arguments. Hence, we have left them out of the implementation altogether (see footnote 29, p.58).

An obvious lacuna in Seuren’s formalization is the treatment of bare nominal elements like pronouns and proper names. This is not surprising, since the theory of Semantic Syntax does not aspire to cover NP-internal syntax extensively as yet. Following on this, we have only incorporated a provisional treatment of these bare nominals into the implementation (see section 3.2.9).

A somewhat different adjustment to the model of Semantic Syntax has been made with respect to the passive construction and its by-phrase. We did not discover a problem within the formalization per se, but rather a technical difficulty within the implementation related to the particular programming tool used. In solving this, we have established that the relative order of PASS and PASSP (PASS<PASSP within the formalization, PASSP<PASS within the implementation) is theoretically neutral (see section 3.2.8). Note that this is an arbitrary choice, so ideally one should have a notation in which this choice need not be made. Such a notation is not offered by GRAMGEN, however.

\textit{Notation}

We have also improved the formalization with respect to its notation. As I have already explained (see section 2.3.4), the technolinguistic notion of formalization (as opposed to theory and implementation) should not be equated with its notation. The \textit{formalization} that Seuren has chosen for his theory of the SA-format is the well-
known mechanism of context-free rewrite rules, which he calls formation rules. His notation, however, is highly idiosyncratic with complex symbols augmented with numerous super- and subscripts – giving it a “somewhat forbidding appearance”, as Seuren himself admits (1996, p.92).

We have translated this idiosyncratic notation into the more generally used notation of EAG (Extended Affix Grammar) without changing the essence of the formalization. The verb translate is appropriate here, because it clarifies the difference between formalization and notation: just as a theory can be expressed in any natural language (e.g. English) and translated into another language (e.g. Dutch) without changing the theory itself, a formalization can be translated from one notation into another without changing the formalization itself. Our EAG notation has various advantages over Seuren’s notation:

- It is more widely used, which facilitates comparing the model of Semantic Syntax to other syntactic models.
- The affix names and values that we have chosen are largely in keeping with more or less accepted terminology, which makes the rules more accessible to anyone unfamiliar with the model of Semantic Syntax.
- The formal properties of EAG’s are well defined and as such, the EAG formalism constitutes a more desirable notational apparatus from a computational point of view.
- The two-layer EAG formalism has made it possible for us to make explicit some of Seuren’s implicit generalizations.

An important manifestation of the third advantage is our treatment of Seuren’s features [PASS] and [DEX]. In Seuren’s formalization, these features differ fundamentally from the other super- and subscripts used. They invoke their own inheritance mechanism by which they are passed down to lower tree levels – a mechanism of which it is not immediately clear whether it extends the formal power of the notational apparatus. In the implementation, however, these features turn out to fit perfectly into the EAG’s computationally well-defined affix system (see section 3.2.8).

With respect to the fourth advantage, the most important generalization we have brought out in this way is the fact that in each S-expansion, the predicate V is always the head. Seuren does not express this explicitly in his formation rules, but in our EAG version the affixes make this crystal clear: the affix of V is always identical to the affix of its mother-S (see section 3.2.1).

Another generalization we have been able to make thanks to the EAG formalism is the insight that at SA level there is no essential difference between adverbs and adverbal prepositions. They occur at all the same levels, whereas the only difference is that the prepositions not only have a subject S but also an object NP. Since the selection of an object NP is easily accounted for by an (independently motivated)
affix, we have done away with the type/class difference between adverbs and prepositions completely (see section 3.2.5).

In order to express even more generalizations, we have added to the standard EAG formalism a special hierarchy notation. With this notation we have been able to make the hierarchy of operators within the auxiliary system explicit and unambiguous, including both the obligatory and the optional operators. It has made it possible for us to combine a number of Seuren’s rules (eight in English, six in Dutch and five in German; see also the next section) into only two EAG rules: a core auxiliary system rule, plus a rule for the quantifiers (see section 3.2.10). Furthermore, the use of entry points within the hierarchy has served to make explicit what type of S can occur as a complement-S (see section 3.2.2). Note that, although the hierarchy notation has weakened the first of the above-mentioned advantages, it has made the fourth advantage a lot stronger and has not affected the other two. More specifically, it has not affected the mathematical properties of the EAG in that it has left unchanged the number of languages that are defined by it.

With respect to the lexicon, we have also made some improvements to the notation (see section 3.3). I have argued that at SA level the term class should be preferred over Seuren’s term category. Also, we have found no reason for splitting the lexicon up into two separate parts for “lexical” and “higher” predicates, like Seuren does. Most importantly, we have translated various symbols, features and restrictions of the Semantic Syntax model into meaningful affix terminals, which we have grouped into affixes needed by the formation rules and those needed by the transformational grammar. Thus, it is now clearer which of the two modules uses which lexical information.

### 3.4.2 Crosslinguistic adequacy

The formation rules have been implemented separately for each language, but it is clear that the similarities between the three rule systems are much larger than the differences. Let us take a brief look at it from a quantitative point of view:

- Only 4 of 22 affix variables are not shared by all three languages – one variable is unique to English (attr/adv1), two are shared by English and Dutch (pass/dex and dex), and one is shared by Dutch and German (adv1/2).

- The type hierarchy is identical for all three languages, except for the placement of the entry points [FIN] (after ATTR and ADV1 in English, but in between ATTR and ADV1 in Dutch and German) and [LEX] (before ADV3 and PASSP/PASS in English, but in between PASSP/PASS and ADV3 in Dutch and German): also, compared to the other two languages, the English hierarchy has one unique member (PROGR) and the German hierarchy lacks one member (DEX).
• Of the 37 grammar rules, 5 only apply to the English and Dutch situations (one for the middle auxiliaries, and four for the external dative); the remaining 32 rules are identical in all three languages, except for the \texttt{dex-variable} in English and Dutch, and the \texttt{adv1/2-variable} in Dutch and German.

Given this limited number of differences, parametrization of the formation rules should be relatively easy. However, we have not carried this through in the implementation because our program tool \texttt{GRAMGEN} provides no turnkey means for parametrization.

A close look at the enumeration above also tells us something about the relationship between the three languages within the formation rules: in some respects English clearly stands apart from Dutch and German, and in others German is the exception to the pattern of English and Dutch. Not one rule or variable is unique to Dutch: it shares with English the external dative and the middle auxiliary, and it shares with German the adverbial levels and the absence of a progressive form.

In Seuren's formalization, the number of differences is larger. We have eliminated them in a number of cases. For example, Seuren's formation rules for English and German differ in that the first allow a \texttt{V\textsubscript{M}} in between the tense operators, whereas the second allow a \texttt{V\textsubscript{FUT}} in the same position. The theoretical assumption, however, is that we are dealing here with one and the same position primarily reserved for the expression of futuricity, but secondarily capable of attracting other modals as well (Seuren 1996, p.85-86). Richter (2000) shows that this is not only the case in English, but also in German. Thus, by following Seuren's theory instead of his formalization and incorporating Richter's findings, we have eliminated a difference between the two languages: in the implementation, both types of predicates are represented as \texttt{V\langle MODAL \rangle} (see section 3.2.3).

A related improvement within the formation rules is our recategorization of the English and Dutch middle auxiliaries. At a theoretical level, Seuren argues that these verbs are "reminiscent of, though not identical to, the modals" (1996, p.138). But in the formalization, Seuren treats the two types of verbs differently. Implementing the middle auxiliaries according to this formalization, however, turned out to pose problems, which we have solved by returning to the theoretical assumption that the middle auxiliaries are a special type of modals. Again, we have deviated from the formalization but not from the theory, and at the same time brought the languages more in line with each other. Now, all three languages have a \texttt{V\langle MODAL \rangle}. A drawback of our recategorization is that the English grammar now wrongly allows for a progressive below a middle auxiliary (see footnote 27, p.49).

Another area where we have brought the formation rules for the three languages more in line with each other is that of the adverbial levels. Implementing the entry points has brought to light a problem with the adverbial level 2 in English. Our solution was to lower Seuren's split level down to below the first tense operator. Not only did this solve the entry point problem, but it also brought the English rules more in line with the Dutch and German ones, which have the level 2 adverbials
below the first tense operator as well. For these languages, too, we have split the level up in a pre- and post-modal level.
CHAPTER 4  The transformational grammar for English

4.1  Introduction

The transformational grammar is implemented in GRAMTSY, a tool especially designed for interpreting such grammars (see Coppen 1991b). As can be seen in Figure 4.1, which is again an excerpt from Figure 2.1 on p.27, a number of files are involved. Relevant to the linguistic user are the “universal” grammar file and rule file, plus the individual parameter files for each of the three languages. These files are compiled into language-specific grammar and rule files. The language-specific files are then combined into a file that can be interpreted by GRAMTSY, to transform the SA into an SS. In the present chapter, I discuss the grammar and rule files for English. In the next chapter, I will show what the universal grammars and rules and the parameters must look like to derive both the English files and the Dutch and German files from them.

The transformational rules of Seuren (1996) are implemented almost entirely one-to-one, i.e. each rule in Seuren’s grammar is one rule in the implementation. As will be shown below, a few minor deviations to this give rise to an interesting re-evaluation of some aspects of the formalization.

About the transformational rules, Seuren says:

“The T-rules should be divided into two classes, the cyclic and the postcyclic rules. The cyclic rules are to be applied cyclically through the tree, starting with the most deeply embedded S and climbing up through successive Ss until the top-S has been dealt with.” (Seuren 1996, p.9)

Furthermore: “A Precycle component is probably useful for certain language-specific adjustments in SA-trees.” (ibid. p.103). So, we implement the transformational grammar as consisting of three subgrammars: the Precycle, the Cycle, and the Postcycle:

%Grammar: Main
  -- Grammar: Precycle
    -- Grammar: Cycle
      -- Grammar: Postcycle
      -- Stop

(Cyclic,BottomUp,RightLeft)

Only the Cycle is applied cyclically, with the cycles being treated bottom up and from right to left; the other two grammars apply non-cyclically, to the whole tree structure at once. I will now discuss the rules of each of these three subgrammars.
I will illustrate the effects of every rule by means of a particular example structure, which will be followed throughout its transformational adventures. The example structure is the SA of sentence (39).

(39) John seemed to have kissed Mary.

The output of the formation rules is the input for the transformational grammar:

---

The output of the formation rules contains several surplus affixes which will not be needed in the transformational grammar. This hold for affixes like NOPERF in V<PERF, NOPERF>, PERF in V<LEX, PERF>, and NP (twice) and PASS on V<LEX, NP, NP, PASS>. These affixes will be removed by a cosmetic rule in the Precycle called Affix-to-Feature, which also replaces the other affixes by feature bundles (see section 4.2.2).
This structure is equivalent to the SA in Figure 4.2, which is how Seuren would depict it.

4.2 Precycle

Although Seuren (1996) stipulates the existence of a Precycle, this component is not formalized. The only thing he explicitly mentions as being part of the Precycle is a transformation called Neg-Raising (Seuren 1996, p.114). This rule is not spelled out, however. Nor is it implemented.

For various mechanisms and processes that Seuren implicitly or explicitly assumes in the theory, the precyclic component turns out to be the best location in the implementation – simply because they need to be carried out before the Cycle starts to operate. Hence we implement various kinds of tree editing rules as part of the Precycle.
Cc: these rules have in common that they are not really transformations, in the sense that they do not change the constituent structure of the tree but only make adjustments to the features. Additionally, the implemented Precycle holds a filter that applies to the input structures:

%Grammar: Precycle
  --> Rule: SA-filter
  --> Grammar: Precyclic-Tree-Editing
  --> Stop

%Grammar: Precyclic-Tree-Editing
  --> Rule: Affix-to-feature-bundle
  --> Rule: S-type
  --> Rule: NP-pruning
  --> Rule: Argument-Functions
  --> Rule: New-Sentence-Unit
  --> Rule: Rest-of-Sentence-Unit
  --> Rule: Spine
  --> Rule: Lookup-Rule-Features
  --> Rule: Perfective-Auxiliary
  --> Stop

4.2.1 SA Filter

Seuren defines two filters on SA structures. He describes the PPI Filter as a mechanism that prevents positive polarity items such as not and EMPH “from standing immediately under the adverbial predicate ‘not’ in the SA of any sentence” (Seuren 1996, p.114). For example, sentence (40b) is ungrammatical because of the double negation and hence it is filtered out.

(40) a John didn’t seem to have kissed Mary.
    b * John didn’t not seem to have kissed Mary.

Obviously, this is only a first approximation of a proper treatment of the notoriously complex phenomenon of polarity items. As Seuren recognizes, the filter is far from categorical:

“The PPI-filter can be overridden when the negation is used metalinguistically, in particular when it cancels presuppositions (the radical negation). The defeasibility conditions of the filter are not worked out here: it is treated as if it were categorical.” (ibid. p.114)

Furthermore, not and EMPH are of course not the only positive polarity items: and not is probably not the only predicate to prohibit a PPI. Also, it remains to be seen whether it is indeed necessary for the filter to apply that nothing stands in between not and the PPI (see also Baker 1970). In other words, the filter may not be as local as Seuren assumes. However, bearing the complexity of polarity facts in mind, we decide to keep to Seuren’s simplified treatment as closely as possible. With Seuren, we leave most of the details unspecified for the time being.

The EMPH filter is somewhat similar: it stipulates that the level 2 adverbial EMPH cannot be followed immediately by another level 2 adverbial, although the formation rules expanding $S^{ADV2}$ and $S^{ADV3}$ normally are recursive (see Seuren 1996, p.114).
The filter excludes sentences like (41b), in which the adverbial element EMPH (surfacing as heavy accented do-support) is followed by a second level 2 adverbial, always. That the reverse order is grammatical is illustrated in (41c).38

(41) a John always seemed to have kissed Mary.
    b * John DID always seem to have kissed Mary.
    c John always DID seem to have kissed Mary.

The main question from our technolinguistic point of view pertains to the formal status of the two filters as such. Seuren does not formalize the filters. He takes them to apply at the SA-level, but neither integrates the filters into the formation rules (although these are supposed to exhaustively define all SAs) nor explicitly defines them as belonging to the transformational grammar. These are really the only two options we have in the implementation, and we decide on the second one.

Although conceptually the filters are not real transformations in the sense of transforming one tree structure into another, they are most naturally accommodated in the transformational grammar. First of all, we do not want to complicate the formation rules with the rather ad hoc and lexeme-specific conditions of the two filters. Integrating the filters into the formation rules would affect the core hyperrule of the auxiliary system, making it significantly more complex. The filters would not be identifiable as such, but rather be intertwined with other details and conditions. Not so in the transformational grammar, in which the two filters can be implemented as one separate simple rule. Should someone ever want to implement new and improved insights concerning positive polarity items, it would be immediately clear where to make the adjustments. Furthermore, our transformational tool GRAMTSY offers an elegant mechanism to implement filters: we just replace the SC part of the rule by a line called FILTER, optionally accompanied by a statement to be conveyed to the user (on screen or in the trace file).

The two filters are highly similar and seem to be two instances of a more general condition on SA structure. Hence, we implement them as one rule:

%Rule: SA-Filter
--> SD: (S ... (V<ADV2> ===1) (S<ADV2> ...2)
 --> COND: ({x1: "not") & {x2: (V<ADV2> {"not" | "EMPH"})}) | {x1: "EMPH")
 --> FILTER: Tree filtered out because of level 2 adverb below &x1

The filter looks for a sentence (S) with an adverbial of level 2 ((V<ADV2> ===)) filled by not or EMPH, respectively. If immediately below this level 2 adverbial there is a positive polarity item ((V<ADV2> {"not" | "EMPH"}) or any level 2 adverbial clause (S<ADV2>), the filter applies. The result is that the tree under

38 These examples may wrongly suggest that the surface order of adverbials is always identical to their SA order. How the exact surface order comes about will become clear in section 4.4.3.
consideration is deleted and a filter message (Tree filtered out because of level 2 adverb below &x1) is written to the output file.

For example, the SA below, corresponding to sentence (40b), would be filtered out by this rule:

4.2.2 Affix-to-feature-bundle and S type

To adjust the simple, one-layered affixes delivered by GRAMGEN to the more complex, multi-layered feature structure that GRAMTSY can handle, we include the following cosmetic rule in the implementation:

%Rule: Affix-to-feature-bundle
--> SD: ... # (S<type>)1 # (V ...) #
--> COND: f1 first_feature_to f2
--> SC: #1 #2 #3 ==> #1 #2!<type<&f2>> #3!<type<&f2>>

This rule looks up the first affix of an S-node and makes it the value of a feature bundle <type> of both the S and its daughter V. Recall that the first affix is always the one belonging to the type/tense-hierarchy within the formation rules in GRAMGEN (see section 3.2.1). At the same time, this rule removes all other affixes with S and V (an exclamation mark in the SC-part of the rule means "replace the feature bundle with this one", see Coppen 1991b). The rule keeps applying until all S-nodes have been processed.

37 For reasons of space, I will leave irrelevant features unspecified within the tree structures.
In the Cycle, it will occasionally be necessary to know whether an S is the very top node of the sentence tree or not (see section 4.3.2 on Lowering). The present rule seems to be the most natural one for having it (as a side effect) add a top feature to the highest S-node:

\[
\begin{align*}
\text{SD: } & \ldots 1 \ (S<^{type} 2 \ # (V \ldots)) \ # \\
\text{COND: } & f_2 \ \text{first_feature_to} \ f_3 \\
\text{SC: } & \ # 1 \ # 2 \ # 3 \\
\Rightarrow & \ [^\langle x_1 \ \text{is_empty} \rangle] \ # 1 \ # 2!\langle \text{type} \& f_3 \rangle \ # 3!\langle \text{type} \& f_3 \rangle \\
\Rightarrow & \ [x_1 \ \text{is_empty}] \ # 1 \ # 2!\langle \text{TOP, type} \& f_3 \rangle \ # 3!\langle \text{type} \& f_3 \rangle
\end{align*}
\]

The effect of this rule on our example structure is the following:

\[
\begin{align*}
S<\text{FIN}> & \quad S<\text{PERF}> \\
| & | \\
V<\text{FIN}> & S<\text{LEX}> \\
| & | \\
V<\text{PERF}, \text{NOPERF}> & S<\text{PERF}> \\
| & | \\
V<\text{LEX, PERF}> & S<\text{PERF}> \\
| & | \\
V<\text{PERF}> & S<\text{LEX}> \\
| & | \\
V<\text{LEX, NP, NP, PASS}> & \text{NP} \ \text{NP} \\
| & | \\
\text{V} & \text{V} \\
| & | \\
\text{PAST} & \text{SIM} & \text{seem} & \text{PREC} & \text{kiss} & \text{John Mary}
\end{align*}
\]

Affix-to-feature-bundle (5x) \[\Rightarrow\]

\[
\begin{align*}
S<\text{TOP, type} & \text{FIN}> \\
| & | \\
V<\text{type} & \text{FIN}> & S<\text{type} & \text{PERF}> \\
| & | \\
V<\text{type} & \text{PERF}> & S<\text{type} & \text{LEX}> \\
| & | \\
V<\text{type} & \text{LEX}> & S<\text{type} & \text{PERF}> \\
| & | \\
V<\text{type} & \text{PERF}> & S<\text{type} & \text{LEX}> \\
| & | \\
V<\text{type} & \text{LEX}> & \text{NP} \ \text{NP} \\
| & | \\
\text{V} & \text{V} \\
| & | \\
\text{PAST} & \text{SIM} & \text{seem} & \text{PREC} & \text{kiss} & \text{John Mary}
\end{align*}
\]

A related rule that is needed in the implementation is one we call S-Type. Recall that in Seuren's formalization, every S-node belongs to one of three (super-)types: it is either $S_0$, $S'$ or $S''$, depending on the number of tense operators it contains (LEX, PERF or FIN, respectively; again, see section 3.2.1). A second superscript specifies whether the S-node belongs to a specific subtype, e.g. $S_0^{\text{PASS}}$, $S_{1\text{ADV2}}$ or $S_{1\text{ATTR}}$. In the implementation, these subtypes are integrated into the type hierarchy in such a way that one cannot tell for an isolated subtype-S to what supertype it belongs. For example, both $S_{1\text{ADV2}}$ and $S_{1\text{ATTR}}$ are implemented as $S_{\text{ADV2}}$. However, some of the transformational rules are sensitive to the supertype of an S-node, so we include a rule to add a supertype feature to each subtype-S:
This rule looks for an $S$-node that does not belong to any of the three supertypes ($S<$type$>$LEX, ^PERF, ^FIN$>$). The type of its daughter-$S$ is looked up ((S$type$>1> ...)) and if this is one of the three supertypes ($f1$ features "<LEX|PERF|FIN>"), it replaces the subtype feature of the mother-$S$ (#2<!type<$f1$>>). The rule applies as many times as necessary to supply all $S$-nodes with the correct supertype feature. Note that the subtype information always remains available as a feature of the $V$-node.

The rule does not apply to the example structure above, since this structure does not contain any subtype $S$-nodes. It has the following effect on the tree structure belonging to sentence (42), for example:

(42) John may not have kissed Mary.
4.2.3 NP-pruning

As I explained in section 3.2.9, the internal syntax of NPs has not yet been treated thoroughly within Semantic Syntax. Seuren (1996) has left several aspects open for further study. In the implementation, we sometimes have to use provisional strategies for the incorporation of nominal elements into the rule system. The precyclic rule NP-pruning is an instance of such a strategy.

Recall that, for the time being, we have been generating bare nominals as V-nodes under non-branching NPs (see section 3.2.9). By doing so, we were able to treat pronouns and proper nouns as Vs, just like all other lexical items. Awaiting a more principled account of NP syntax, however, we want to keep our NPs as simple as possible and in line with Seuren’s analysis, so we now assume a precyclic rule to remove the intermediary V-node from within the non-branching NP:

%Rule: NP-pruning
  --> SD: (S ... (NP # (V<Pron> # ... #) #)
  --> COND: None
  --> SC: #1 #2 #3 #4 ==> #1 #3

The effect of this rule on our example structure is straightforward:

```
%Rule: NP-pruning
  --> SD: (S ... (NP # (V<Pron> # ... #) #)
  --> COND: None
  --> SC: #1 #2 #3 #4 ==> #1 #3

The effect of this rule on our example structure is straightforward:

```

```

<table>
<thead>
<tr>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

PAST SIM seem PREC kiss John Mary

NP-pruning (2x) ==> 

<table>
<thead>
<tr>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

PAST SIM seem PREC kiss John Mary
```
4.2.4 Argument Functions

Seuren describes a procedure to determine the argument function (AF) of constituents, i.e. whether they are subject (SU), direct object (DO) or indirect object (IO):

“The AFs SU, DO or IO are based primarily on the number and position of the argument terms. That is, no parameters other than connections are needed for the definition of AFs: in VSO structures, the first NP or S after V is subject; the last NP or S is direct object provided there is also a subject; any NP between subject and direct object is the indirect object. This is the default definition of AFs.”

(Seuren 1996, p.49)

Since these argument functions are used in some of the transformational rules of the Cycle, we implement a precyclic rule that encodes them as features on the arguments. The basic rule looks like this:

%Rule: Argument-Functions
   --> SD: . . . (S (V . . .) #
         (\ARG . . .) # ((NP . . .)?1 (\ARG . . .)2)? #
   --> COND: (^x2 is empty) -> x2 plus "<af<DO>>" &
            (^x1 is empty) -> x1 plus "<af<IO>>"
   --> SC: #1 #2 #3 ==> #1 #2<af<SU>> &x1 &x2

This rule looks for a predicate (V), followed by an S or NP (\ARG); there may also be another S or NP (((\ARG . . .)2)? and if so, an NP may stand in between these two ((NP . . .)?1). The first argument is assigned its feature bundle in the SC (#2<af<SU>>). If the optional arguments are present (^x2 is empty) and ^x1 is empty), they are assigned their appropriate feature bundles in the condition part of the rule (x2 plus "<af<DO>>" and x1 plus "<af<IO>>").

However, a complication arises with this rule. Some predicates have deviant argument frames: “a term may be assigned an AF feature on lexical grounds” (Seuren 1996, p.49). Such argument function deviations are listed in the lexicon with the predicate. In English, in some cases, the middle argument is the direct object, rather than indirect object (for example, with the verb help the second argument is the direct object and the third is an embedded clause as in I helped John to find his mother). We extend the rule as follows:

---

38 At first sight, it may seem more logical to assign the argument functions in the formation rules, since that is where the argument frame is looked up in the lexicon. It would indeed be feasible for the standard argument functions to do so. However, we have two reasons for postponing the assignment to the transformational grammar. First, the deviant argument functions which are listed in the lexicon (see below), are easier to assign within the transformational apparatus. And secondly, the transformational rule can be put to use twice (by re-calling it within the Cycle; see section 5.5.1) since the argument functions may change due to the application of cyclic transformational rules such as Predicate Raising.
Now, if the lexicon lists a deviant argument function ({{?&LEX: x1}: (af<DO>>)), the middle argument is assigned that function instead of the default one (x2 plus "!af<DO>>") and the third argument is stripped of its function (x3 plus "^af>").

The result of the rule Argument-Functions is illustrated below:

4.2.5 New Sentence Unit and Rest of Sentence Unit

Seuren defines a notion of sentence unit:

"Each nucleus with its higher auxiliary material forms a sentence unit. A main clause thus forms a sentence unit down to any possible embedded complement clause, which starts a new sentence unit. For convenience we mark successive sentence units with subscripts. (...) If there is no complement-S the numerical subscript is not used."

(Seuren 1996, p.79)

Although Seuren's numerical subscripts are only used for convenience and not essential for further transformational treatment of the sentence, we implement them as
well. We translate them into feature bundles with a numerical value, and define two precyclic rules to assign these features. First, there is a rule to determine whether the tree contains a complement-S at all; if it does, the top node of each clause receives its correct unit feature. A second rule copies this unit feature to all S-nodes within the same clause.  

%Rule: New-Sentence-Unit

\[ \rightarrow SD: (S \ldots (S^{type \text{LEX}}) \ldots) \iff \{NP \? \} \# (S^{\text{unit}}) \# \]
\[ \rightarrow COND: \text{None} \]
\[ \rightarrow SC: \#1 \#2 \implies \#1^{unit \leq 0} \#2^{unit<i+i>} \]

%Rule: Rest-of-Sentence-Unit

\[ \rightarrow SD: \ldots (S^{\text{unit}<1}) \iff \{NP \iff \} \# (S^{\text{unit}}) \# \]
\[ \rightarrow COND: \text{None} \]
\[ \rightarrow SC: \#1 \#2 \implies \#1 \#2^{unit<\&fl>} \]

The rule New-Sentence-Unit looks for a sentential argument of a lexical predicate (V^{LEX}). If this argument has no unit feature yet (S^{\text{unit}}), it gets assigned such a feature with a cumulative value, i.e. the previous value plus one (#2^{unit<i+i>}). The top S-node receives the unit feature with value 0 (#1^{unit<0}>). Next, the rule Rest-of-Sentence-Unit looks for an S-node with a unit feature (S^{\text{unit}}). All S-nodes that belong to the same unit (S^{\text{unit}}) receive the same unit value (#2^{unit<\&fl>}).

Note that in both rules, an optional NP-node is allowed in between the two S-nodes. This concerns two different situations. On the one hand, a new sentence unit can start directly under an NP-node because with some lexical predicates the complement-S may be headed by an NP-node, as discussed earlier (see Figure 3.13 in section 3.2.9). On the other hand, a sentence unit may continue under an NP-node in the case of quantifiers (see Figure 3.14 in section 3.2.10).

These rules apply to our example structure in the following manner:

```
New-Sentence-Unit, Rest-of-Sentence-Units (3x) \implies
```

---

39 Although it is possible in principle to integrate the two assignment steps into one rule, we have chosen not to do so because it would become a fairly complex rule with an extremely intricate COND-part. The present two rules are relatively simple and easy to understand for any future user of the generator.
4.2.6 Spine

The notion of spine plays a central role in Semantic Syntax. Many rules of the Cycle are sensitive to a node being on the spine or not.

"Spines are used to indicate the directionality of an expansion: if an expansion is right-branching the spine is on the right, and analogously for left-branching expansions. In lexical expansions with a complement-S the spine connects the dominating S-node with that S. If there is no complement-S there is no spine."

(Seuren 1996, p.74)

Seuren represents the spine as heavy lines in the tree structure, as illustrated in Figure 4.3. The question is, of course, how to represent the spine in the implementation. Since in a structure of labelled brackets the connecting lines are not elements that can be manipulated, we need to find an alternative representation. The most natural solution seems to be a feature on the lowest of the two nodes connected by a spine. This makes it unambiguously clear, also in each local tree, which node is on the spine and which is not.

The following rule adds the relevant feature to the S-nodes:

%Rule: Spine
--> SD: ... (S <= # (S<^spine> ...) #
--> COND: None
--> SC: #1 #2 ==> #1 #2<spine>

This rule looks for any S-node with a mother S ((S <= (S ...)); if this node does not have a spine feature yet (S<^spine>), it receives one (#2<spine>).

It should be noted, however, that Seuren's formal definition of spine is in fact somewhat more complicated:

"The notion spine can now be defined as follows (where α stands for left or right; the directionality (α-branching) of a node N may change as a result of SURFACE CATEGORY ASSIGNMENT induced by the rule that led to the expansion in question): A spine is a connection between a parent node P and a dependent node D satisfying either of the following conditions:
(a) P.cat=[S or /S] and D.cat=[S or /S], or
(b) P has a binary expansion and one of its dependents, B, has been adopted and was set for α-directionality before the adoption, and D is on the α-side of the expansion."

(Seuren 1996, p.78)
Figure 4.3 The spine represented as a heavy line.

Our rule only implements situation a; the b-case is integrated into rules where adoption takes place (as discussed in section 4.3).

The effect of the rule Spine is straightforward:
4.2.7 Lookup Rule Features

The rules of the Cycle are all induced by rule features on the predicate. These features are stored in the lexicon, from which they are retrieved by the following precyclic rule:

\[
\text{%Rule: Lookup-Rule-Features} \\
\rightarrow \text{SD: \ldots # (V<^\text{rulf}> \equiv 1) #} \\
\rightarrow \text{COND: \{}(?&LEX: x1): (V<^\text{rulf}> \equiv 2) \& (^f2 \text{ is_empty_feature}) \} \\
\rightarrow \text{SC: #1 #2 \Rightarrow #1 #2<rulf<&f2>}
\]

Each lexical predicate that does not have any rule features yet (\(\text{((V<^\text{rulf}> \equiv =)}\)) is looked up in the lexicon to see whether one or more rule features are listed for it. If that is the case, these features are copied onto the predicate (#2<rulf<&f2>>).

The rule applies as follows:

\[
\text{S} \\
\text{V} \quad \text{S} \\
\text{V} \quad \text{S} \\
\text{V} \quad \text{S} \\
\text{V} \quad \text{NP} \quad \text{NP} \\
\text{PAST} \quad \text{SIM} \quad \text{seem} \quad \text{PREC} \quad \text{kiss} \quad \text{John} \quad \text{Mary}
\]

\[
\text{Lookup-Rule-Features (4x) \Rightarrow} \\
\text{S} \\
\text{V<rulf<GR,L<\text{V}>>>} \\
\text{V<rulf<GR>>>} \\
\text{V<rulf<<>>} \\
\text{V<rulf<GR,L<\text{V}>>>} \\
\text{V<rulf<PA>,L<\text{V}>>>} \\
\text{S} \\
\text{V} \quad \text{NP} \quad \text{NP} \\
\text{PAST} \quad \text{SIM} \quad \text{seem} \quad \text{PREC} \quad \text{kiss} \quad \text{John} \quad \text{Mary}
\]

4.2.8 Perfective Auxiliary

In section 3.2.1, I explained that we have chosen to use an abstract filler for the perfective tense operator, rather than a lexical filler. The abstract filler \text{PREC} ("preceeding") is to result in a perfect tense and thus has to be replaced by a perfective auxiliary at some stage in the generation process.
We implement this replacement operation as a precyclic rule, so as to make the tree easier to read during the rest of the generation process (there is no fundamental reason for this choice – the replacement could be done just as well in the Postcycle, for example). For English, this is very simple:

%Rule: Perfective-Auxiliary  
---> SD: (S ... (V<type<PERF>> # "PREC" # )  
---> COND: None  
---> SC: #1 #2 ==> #1 "have"

This rule looks for a perfective operator (V<type<PERF>>) with the terminal "PREC" as its filler, and replaces this filler by the terminal "have". The rule has the following effect on our example structure:

![Diagram of tree structure]

Perfective-Auxiliary ==> 

![Diagram of modified tree structure]

4.3 Cycle

Before I go into the details of the subgrammars and rules of the Cycle, a few more general remarks should be made about the relationship between Seuren’s theory and formalization on the one hand and our implementation on the other. The transformational component of Semantic Syntax consists of various operations or routines, which Seuren classifies in a number of ways. First, a routine is classified as either local (applicable at a certain stage in the transformational process) or global (applicable anywhere, i.e. non-local). Local routines are what one usually calls rules:
global routines can be seen as conventions or standard procedures. However, for reasons of efficiency, these standard procedures are not implemented as true global routines: their applicability is not tested at every stage of the transformational process, but only at those moments when other routines may have created a situation in which they are applicable.

A second classification criterion that Seuren uses is the way in which a routine is induced: procedurally, structurally, by a category, or lexically. PROCEDURALLY induced routines are applicable within a larger routine — that is, as a subroutine. Depending on their complexity and frequency, we either integrate these routines into the larger routines as true subroutines (for example, Adoption and Downgrading), or realize them as separate rules (for example, Surface Category Assignment and Tree Pruning). Furthermore, there seems to be no real ground for the distinction between structurally induced routines and category-induced routines in the implementation. For in the GRAMTSY formalism, all rules are essentially structurally induced since it is the Structural Description that (together with the Conditions) defines whether a transformation should be applied or not. In the remainder of this chapter, it will turn out that the applicability of the so-called category-induced rules like Copula Insertion does not only depend on the presence of a category, but is just as sensitive to some wider structure as the structurally induced rules. We will therefore regard them as structurally induced as well. Only the lexically induced rules clearly stand apart from the other rules, since they are induced by rule features that are retrieved from the lexicon (see the description of the precyclic rule Lookup Rule Feature, p.89-89). They also apply before all the other rules.\footnote{50}

Thus we come to the following division of the Cycle in subgrammars:

```
%Grammar: Cycle
   --> Grammar: Lexically-Induced
   --> Grammar: Structurally-Induced
   --> Grammar: Corollaries
   --> Stop
```

The corollaries are also assumed by Seuren:

"It has proved necessary, for the languages that have been investigated, to formulate a few corollaries between the Cycle and the Postcycle. These are to do with categorial relabellings, the default insertion of particles or complementizers, the delimitation of the AUX-area in V-clusters and similar operations. Since these can be called 'transformational' only in a weak sense they have not been included among the postcyclic transformations." (Seuren 1996, p.23)

The difference is that Seuren seems to regard the corollaries as non-cyclic routines whereas in the implementation they are applied cyclically. It appears to be perfectly well possible for most of the corollaries to implement them as cyclic rules, and the advantage of doing so is that the subgrammar Corollaries will now also offer room for procedurally induced routines that cannot be easily integrated into larger rou-

\footnote{50 But see also footnotes 49 and 50 on p.110 and p.112, respectively.}
times, such as Tree Pruning. The only corollary that cannot be implemented as a cyclic rule is AUX-delimitation; this becomes the first rule of the Postcycle.

About the rule ordering within the Cycle, Seuren says:

"When more than one rule is required or allowed for by the same predicate on the same cycle, as for example the rule pair <SR,L> (SUBJECT RAISING and LOWERING) for PRES or PAST, they are intrinsically ordered with respect to each other: application of one rule must not destroy the structural conditions for application of the other. Here this means that SR is to apply before L." (ibid. p.107)

GRAMTSY forces us to order the rules explicitly. The only grounds for doing so are situations as mentioned by Seuren, when more than one rule is induced by the same predicate: we always need to have the so-called feeding order (or, at least, a non-bleeding order). This gives us the following guidelines:

- Subject Deletion comes before Subject Raising
- Subject Raising comes before Lowering
- Object Incorporation comes before Lowering
- Participle comes before Lowering

The rest of the ordering is arbitrary:

\[
\begin{align*}
\text{%Grammar: Lexically-Induced} & \rightarrow \text{Rule: Subject-Deletion} \quad \text{(Once,,)} \\
& \rightarrow \text{Rule: Subject-Raising} \quad \text{(Once,,)} \\
& \rightarrow \text{Rule: Object-Incorporation} \quad \text{(Once,,)} \\
& \rightarrow \text{Rule: Participle} \quad \text{(Once,,)} \\
& \rightarrow \text{Rule: Lowering-to-S} \quad \text{(Once,,)} \\
& \rightarrow \text{Rule: Lowering-to-V} \quad \text{(Once,,)} \\
& \rightarrow \text{Rule: Lowering-to-the-Right} \quad \text{(Once,,)} \\
& \rightarrow \text{Stop} \quad \text{(Once,,)} \\
\text{%Grammar: Structurally-Induced} & \rightarrow \text{Rule: Extraposition} \quad \text{(Once,,)} \\
& \rightarrow \text{Rule: Copula-Insertion} \quad \text{(Once,,)} \\
& \rightarrow \text{Stop} \quad \text{(Once,,)} \\
\text{%Grammar: Corollaries} & \rightarrow \text{Rule: Grammar: Tree-Pruning} \quad \text{(Once,,)} \\
& \rightarrow \text{Rule: Particle-Insertion} \quad \text{(Once,,)} \\
& \rightarrow \text{Rule: Surface-Category-Assignment} \quad \text{(Once,,)} \\
& \rightarrow \text{Rule: That-Insertion} \quad \text{(Once,,)} \\
& \rightarrow \text{Rule: Cyclic-Cosmetics} \quad \text{(Once,,)} \\
& \rightarrow \text{Stop} \quad \text{(Once,,)} \\
\end{align*}
\]

I will discuss these rules in the order in which they occur in the generation of our example sentence. That is, I will begin with rules that apply in the deepest S-cycle.

---

4 Coppen has developed a supplemental tool for GRAMTSY users to abstract away from rule ordering if desired. The tool makes it possible for the user to specify partial ordering statements like "Subject Deletion comes before Subject Raising" and "Subject Raising comes before Lowering", on the basis of which it derives a complete rule ordering. The tool also offers the means to supply the grammars and rules with documentation. We do not use this tool, however.
and work my way up from there. After that, I will discuss the rules that do not apply at all in our particular example structure.

In the deepest S-cycle of the example structure, no rules apply at all – the lexical predicate \textit{kiss} does not induce any cyclic rules. In the next cycle, the predicate \textit{have} first induces the rule \textit{Participle}.

4.3.1 \textit{Participle}

Seuren assumes two very similar cyclic rules: Past Participle and Present Participle. These rules add an abstract affix to verbs below the perfective, passive or progressive auxiliary. Subsequently, in the morphological component, the verb and the affix together will be transformed into the appropriate past or present participle.

The first rule is formally defined as follows:

\begin{quote}
"PAST PARTICIPLE (PaP): The highest lexically filled non-spine V-node in the V-cluster of the argument-S \(\alpha\)-adopts \textit{Aff}[EN]. With postcyclic AFFIX HANDLING the V-node directly above \textit{Aff} is relabelled 'PaP'." (Seuren 1996, p.95)
\end{quote}

It may be implemented in a straightforward manner:

\begin{verbatim}
%Rule: Past-Participle
  --> SD: (S # (V<rulf<PAP>> ...) <= (S ... # (V<^spine> ===) #
  --> COND: None
  --> SC: #1 #2 #3
        == #1 #2<rulf<PAP>> "(AFF-EN)" >> #3
\end{verbatim}

This rule looks for a Past Participle inducing predicate ((V<rulf<PAP>> ...)). It traces the highest lexically filled non-spine V-node in the V-cluster of the argument-S ((S ... (V<^spine> ===)). This V-node left-adopts the past participle affix node ("(AFF-EN)" >> #3). The rule-inducing feature is removed from the predicate (\#2<rulf<PAP>>). The second part of Seuren’s definition, concerning the relabelling with Affix Handling, is not expressed in this rule but in the postcyclic rule Affix Handling itself (see p.133-135).

The second rule, Present Participle, is defined and implemented as follows:

\begin{quote}
"PRESENT PARTICIPLE (PrP): The highest lexically filled non-spine V-node in the V-cluster of the argument-S \(\alpha\)-adopts \textit{Aff}[ING]. With postcyclic AFFIX HANDLING the V-node directly above \textit{Aff} is relabelled 'PaP'." (ibid. p.95)
\end{quote}

\begin{verbatim}
%Rule: Present-Participle
  --> SD: (S # (V<rulf<PRP>> ...) <= (S ... # (V<^spine> ===) #
  --> COND: None
  --> SC: #1 #2 #3
        == #1 #2<rulf<PRP>> "(AFF-ING)" >> #3
\end{verbatim}

This rule looks for a Present Participle inducing predicate ((V<rulf<PRP>> ...)). It traces the highest lexically filled non-spine V-node in the V-cluster of the argument-S ((S ... (V<^spine> ===)). This V-node left-adopts the present participle affix node ("(AFF-ING)" >> #3). The rule-inducing feature is re-
moved from the predicate (#2<^rulf<PRP>>). Again, the second part of Seuren’s definition is not expressed in this rule.

Given the similarity of the two rules, we decide to integrate them into one rule called Participle:

%Rule: Participle

\[\text{SD: (S # (V<rulf<>l> ...) <= (S ... # (V<^spine> ===) #}
\]

\[\text{COND: (f1 features "<PAP>" & (x2 = "(AFF-EN)")) |}
\]

\[\text{(f1 features "<PRP>" & (x2 = "(AFF-ING)"))}
\]

\[\text{SC: #1 #2 #3}
\]

\[\text{==> #1 #2<^rulf<PRP,PRP> 6x2 >> #3}
\]

According to the definition of the notion spine (see section 4.2.6), binary expansions that come into being through adoption have a spine. In this case, the adopting V-node gets to be on the spine, so in the SC we need to add the feature <spine> to this node. Also, to prevent the proliferation of features, we remove all features from the newly created mother-V: 42

\[\text{SC: #1 #2 #3}
\]

\[\text{==> #1 #2<^rulf<PRP,PRP> (6x2 >> #3<spine>)<^type,<spine>}
\]

The effect of the rule on the relevant S-cycle of our example structure is:

\[
S \quad S
\]

\[
V<^rulf<PRP,L<V>> \quad V<^rulf<L<V>>
\]

\[
have \quad V \quad NP \quad NP \quad have \quad V<^spine> \quad EN \quad kiss \quad John \quad Mary
\]

\[
\text{Participle} \Rightarrow \quad \text{S}
\]

\[
V<^rulf<PRP,L<V>> \quad S
\]

\[
\text{have} \quad EN \quad kiss \quad John \quad Mary
\]

**4.3.2 Lowering**

One of the central rules of the cycle is Lowering. It is typical of all operators of the auxiliary system, that is, of the tense operators, quantifiers, negation and other sen-

---

42 In fact, Seuren does not say much about the copying of features when adoption takes place. He only says something about the rule features: they are moved from the adopting node to the newly created node. This only occurs with Object Incorporation (see section 4.3.3).
tential modifiers. In general, this rule detaches the inducing predicate and incorporates it into its argument-S. Seuren gives the following formal definition:

“For any node N—cat=V and N—p=Sα, where Sα.d=α-Sα-β or Sα.d=α-\overline{Sα}β or Sα.d=α-np[Sα]-β, and N—rulf=<L> (α, β are possibly null strings of nodes):
(a) DETACH the node N,
(b) RE-ATTACH N in the appropriate position (variable or lexically specified),
(c) apply SURFACE CATEGORY ASSIGNMENT (SCA) if necessary.” (Seuren 1996, p.66)

The landing site varies across predicates. For most predicates, the landing site is specified in the lexicon. There are three options:

- the V-constituent of the argument-S is to adopt the predicate (Lowering to V);
- the argument-S itself is to adopt the predicate (Lowering to S);
- the predicate is to become the rightmost daughter of the Argument-S (Lowering to the right).

If the predicate is a quantifier, however, the landing site is not lexically determined; instead, the quantifier must be lowered onto the (first occurrence of the) variable that is bound by it. Thus, there are four different types of Lowering with respect to the landing site. Let us have a look at them one by one.

Lowering to V is the most common form of Lowering. This rule helps to build up a V-cluster, by adding the auxiliaries, tense affixes and some of the adverbs to the matrix verb. The rule can be implemented as follows:

%Rule: Lowering-to-V
---> SD: (S # (V\<rulf\<L\<V\>) \ldots) # <= (\{(NP )? (S # (V \ldots) #
---> COND: None
---> SC: #1 #2 #3 #4
     ==#1 #3 #2\\<rulf\<L\>> \ldots #4

This rule looks for a predicate marked for Lowering to V ((V\<rulf\<L\<V\>) \ldots)). It traces the V-constituent of the argument-S ((S (V \ldots))), which may or may not be headed by an NP-node (\{(NP )?). The inducing predicate is then left-adopted by this V-constituent and simultaneously the responsible rule feature is removed (#2\\<rulf\<L\>) \ldots #4).

Again, just as with the adoption routine in the rule Participle (see section 4.3.1), the adopting V-node gets to be on the spine. Furthermore, as the definition of Lowering states, a corollary of this rule is a routine called Surface Category Assignment. This routine is a separate rule in the implementation (see section 4.3.4), which will be triggered by a feature we add to the lowered V-node at this stage:

---> SC: #1 #2 #3 #4
     ==#1 #3 #2\\<SCA, \rulf\<L\>> \ldots #4\\<spine, ^\text{type}>

The effect of this rule on the relevant S-cycle is:
A second type of Lowering. Lowering to S, also involves adoption. But in this case, the adoptive node is not the V-constituent of the argument-S but rather the argument-S itself, “thus leaving the structure intact and, where applicable, causing only a category change in virtue of SURFACE CATEGORY ASSIGNMENT” (Seuren 1996, p.66). The rule is implemented as follows:

%Rule: Lowering-to-S
--> SD: (S # (V<rulf<L<S>> ...)) # (NP ...)? {(NP )}? #
(S ...) #
--> COND: None
--> SC: #1 #2 #3 #4
   ==> #1 #3 (#2<SCA, ^rulf<L>> >> #4<spine>)<^spine, ^type>

This rule looks for a predicate marked for Lowering to S ((V<rulf<L<S>> ...)) and traces its argument-S ((S ...)), which again may or may not be headed by an NP-node ((NP )?). The inducing predicate is left-adopted by this S-node (#2 >> #4) and the same feature adjustments apply as with Lowering to V above.

Some Lowering-inducing predicates offer a choice of landing sites: “Disjunctive subscripts, as in L_{s/v} or L_{s/right} imply a choice” (Seuren 1996, p.99). Examples are time adverbials like today and yesterday which may either be lowered to S or to the far right. Thus, sentences (43a) and (43b) represent two surface structures deriving from the same underlying SA.

(43) a  Yesterday John kissed Mary.
     b  John kissed Mary yesterday.

If we place the rule of Lowering-to-S before the other Lowering rules in the implementation, this choice of landing sites can be accounted for as follows:

--> SD: (S # (V<rulf<L<S>1>> ...)) # (NP ...)? {(NP )}? #
(S ...) #
--> SC: #1 #2 #3 #4
   ==> #1 #3 (#2<SCA, ^rulf<L>> >> #4<spine>)<^spine, ^type>
   ==> [fl features "<V|R>"] #1 #2<^rulf<L>S>> #3 #4

If the Lowering feature of the inducing predicate mentions another landing site besides S ((V<rulf<L<S>1>> ...)) with fl features "<V|R>") then the rule offers an alternative SC which leaves the structure intact and only removes the
Lowering-to-S feature (#2<rulf<LS>>>). The remaining Lowering feature will trigger the rule Lowering-to-V or Lowering-to-the-Right.

Lowering onto the S-argument is subject to the condition that it only applies at the very top of the tree. This condition only holds when there is an alternative landing site: "L\_right has the additional condition that the first option, left adoption by S, applies only if the operator to be lowered is at the top of the tree" (Seuren 1996, p.99).\(^{43}\) We implement this "top condition" as follows:

\[
\begin{align*}
\text{SD:} & \quad (S\langle 1 \# (V\langle rulf<LS>2\rangle \ldots) \# (NP \ldots)? (NP)?) \# (S \ldots) \# \\
\text{SC:} & \quad #1 \#2 \#3 \#4 \\
\text{SCA:} & \quad \langle f1 \text{ features } "<\text{TOP}>"\rangle \mid \langle f2 \text{ features } "<V|R>"\rangle \\
& \quad \langle f2 \text{ features } "<\text{TOP}>"\rangle \#1 \#2\langle rulf<LS>>> \#3 \#4
\end{align*}
\]

Now, the actual lowering onto the S-argument only takes place if the mother-S of the predicate to be lowered is at the top of the tree (\(S\langle 1 \text{ with } f1 \text{ features } "<\text{TOP}>"\rangle\))\(^{44}\) or if there is no alternative landing site (\((V\langle rulf<LS>2\rangle \ldots)\) with \(\langle f2 \text{ features } "<V|R>"\rangle\)).

The third variety of Lowering is Lowering to the right. This type of Lowering does not involve adoption. It inserts the inducing predicate directly under its argument-S, as the rightmost daughter.

\%

\text{Rule: Lowering-to-the-Right}
\text{SD:} & \quad (S \# (V\langle rulf<LR>>> \ldots) \# (NP \ldots)? (NP)?) (S \ldots) \# \\
\text{SC:} & \quad #1 \#2 \#3 \#4 \\
\text{SCA:} & \quad \#1 \#3 \#2\langle SCA, rulf<LR>>> \\
\]

This rule looks for a predicate marked for Lowering to the right (\((V\langle rulf<LR>>> \ldots)\)) and traces its argument-S (\((S \ldots)\)), which again may or may not be headed by an NP-node (\((NP)?)\). The inducing predicate is inserted at the far right under this S-node; it is stripped from its rule feature and it receives a feature to trigger Surface Category Assignment (#2<SCA, rulf<LR>>). Note that no spine feature is added, because we are not dealing with adoption here.

Lowering to the right is sensitive to various elements within the argument-S, such as embedded clauses, in that the lowered predicate may halt before them rather than move to the far right. The ordering of adverbs and prepositions with respect to em-

\(^{43}\) Note that twenty pages later Seuren appears to regard it as a general constraint: "Lowering onto S by left-adoption is generally constrained in that it is allowed only at the top of the SA-tree: no higher structure must be present." (p.119). Seuren (personal communication) stresses, however, that the alternative landing site must indeed be present for the condition to hold.

\(^{44}\) Recall that the feature "<TOP>" was added to the highest S-node as a side effect of the precyclic rule Affix-to-feature-bundle (see section 4.2.2). Although this may seem somewhat laborious and ad hoc, a feature is the only way to check if we are indeed at the top of the tree in the cyclic grammar, since the cyclic rules cannot look beyond the S-node of the cycle at hand.
bedded clauses has been studied extensively over the years (see, for example, Ross 1967/1986). What has become clear is that the judgements vary across speakers and are never absolute. In general, prepositional phrases seem to be freer than adverbs, and infinitive clauses seem to be freer than finite clauses. Thus, most people strongly prefer (44a) — with the adverb before the embedded clause — over (44b), whereas (45b) — with the adverb after the embedded clause — seems to be just as acceptable as (45a).

(44)  a  I will tell them tomorrow that John has kissed Mary.
       b  * I will tell them that John has kissed Mary tomorrow.

(45)  a  John has wanted for a long time to kiss Mary.
       b  John has wanted to kiss Mary for a long time.

Setting all the subtle details aside for further study, Seuren (1996) captures the general tendency by stating that, in English, Lowering to the right never crosses an embedded S, and it optionally crosses an embedded /S. This can be implemented as follows:

```plaintext
--> SD: (S # (V<rule<]<L><d>) ..)... # (NP ...)? ((NP )?)
(S ==> # ( ...))1 # ) #
---> SC: #1 #2 #3 #4 #5

==>[ {x1: {NP }? (S<type> ) } | {x1: {NP }? (S<SLASH> )}]
     #1 #3 #2<SCA,^rul<]<L><d> #4 #5

==>[ ^{x1: {NP }? (S<type> )}]
     #1 #3 #4 #2<SCA,^rul<><L><d> #5
```

Now, if the rightmost daughter of the argument-S is an S- or /S-node, possibly headed by an NP-node ([{x1: {NP }? (S<type> )} | {x1: {NP }? (S<SLASH> )}]), the lowered predicate is placed before it. If the rightmost daughter is not an S-node, then the lowered predicate crosses it. Thus, if the rightmost daughter is a /S-node this rule outputs two structures.

Not only embedded clauses may check Lowering to the right. Another element that may cause an early landing is a Prepositional Phrase. Thus we have both (46a) and (46b).

(46)  a  Mary was kissed passionately by John.
       b  Mary was kissed by John passionately.

This can be integrated into the implemented rule as follows:

45 In fact, the order of lowered elements is even more complex if we look at scope-sensitive operators. In the theory, this is accounted for by the so-called Scope Ordering Constraint (Seuren 1996, p.300-309), which, however, we have not implemented yet.
Now, if the rightmost daughter is a PP, the rule again outputs two structures: one with the lowered predicate before the PP and one with the predicate after it.

The fourth variety of Lowering, Lowering to a variable, has not yet been implemented. This type of Lowering is limited to quantifiers (and is therefore also referred to as Quantifier Lowering by Seuren 1996, p.301). The precise effect of and conditions on Lowering to a variable are heavily intertwined with the internal grammar of noun phrases, which is not worked out in detail in the formalization nor in the implementation (see also sections 3.2.9 and 3.2.10).

### 4.3.3 Tree Pruning

A procedurally induced routine is Tree Pruning. It applies at the end of any cyclic or postcyclic routine, to erase superfluous nodes that remain after the routine is carried out. Typically, these superfluous nodes are non-branching. Tree Pruning, too, comes in a number of varieties.

In the implementation, each of the varieties of Tree Pruning constitutes a separate rule. These rules are grouped into a subgrammar, which is called from the higher subgrammar Corollaries. The same subgrammar will be called again at various stages in the Postcycle. We will discuss the rules one by one.

%Grammar: Tree-Pruning

---> Rule: Tree-Pruning-1
---> Rule: Tree-Pruning-2
---> Rule: Tree-Pruning-3
---> Stop

The most common form is Tree Pruning 1. It is formally defined as:

“For any nodes X, Y such that X.cat=C and X.d=Y and Y.cat=C:

(a) ERASURE of X,

(b) RE-ATTACHMENT of Y higher up.”

(Seuren 1996, p.56)

In other words, a non-branching node which directly dominates a node of the same category is erased and the remaining node is re-attached higher up (see Figure 4.4 for a schematic representation). This can be implemented as follows:

%Rule: Tree-Pruning-1

---> SD: ... # (<>1 # (<>11 ...) #) #
---> COND: None
---> SC: #1 #2 #3 #4 ==> #1 #3
This rule looks for a node whose only daughter has the same label as itself \((\langle 1 \ (\& 1 \ \ldots ) \rangle)\). This node, consisting of a labelled opening bracket \((#2)\) and a closing bracket \((#4)\), gets erased.

When the non-branching node that is erased is an S-node, “the receiving S takes on the higher superscript of the erased S-node” (Seuren 1996, p.97). Thus, we extend the SC-part of our rule:

\[ \Rightarrow \text{SC: } #1 \ #2 \ #3 \ #4 \implies #1 \ #3<^{\text{type}},^{\text{af}},^{\text{spine}},^{\text{fl}}> \]

Now, the rule removes the S-features from the remaining node and adds to it the feature bundle of the erased node \((#3<^{\text{type}},^{\text{af}},^{\text{spine}},^{\text{fl}}>)\).

The rule has the following effect on the example tree structure at this stage:

```
S<type<PERF>,af<SU>,unit<1>,spine>
  |  
S<type<LEX>,af<SU>,unit<1>,spine>
  |   
V  V 
  |   
  |   
AFF  V 
  |   
have EN kiss John Mary

Tree-Pruning-1 ==> 

S<type<PERF>,af<SU>,unit<1>,spine>
  |  
V  V 
  |   
  |   
AFF  V 
  |   
have EN kiss John Mary
```

Tree Pruning 2 is very similar to Tree Pruning 1. Formally, it reads:

“For any nodes X, Y, Z such that X.cat=C and Y.cat=C and X.d=Z and X.p=Y:
(a) ERASURE of X,
(b) RE-ATTACHMENT of Z higher up.”

(ibid. p.57)
Thus, any non-branching and not lexically filled node directly dominated by a node of the same category is erased and its dependent node is re-attached higher up (see Figure 4.5). The implementation is also very similar to that of Tree Pruning 1:

```
X₁  ⇒  X₁
     /    /
    X₂   Y

Figure 4.5 Tree Pruning 2.
```

%Rule: Tree-Pruning-2

```
-> SD: ... (<\>1 # (&11<>)2 # ( ... ) # ) #
-> COND: None
-> SC: #1 #2 #3 #4 => #1 #3<^type,^af,^spine,&f2>
```

This rule, too, looks for a node with only one daughter, but now it has to share its label with its mother node ((<\>1 ( &11 ( ... ) ) ). This node, consisting of a labelled opening bracket (#2) and a closing bracket (#4), gets erased. Again, the features of the remaining mother node are replaced by the features of the erased node (#3<^type,^af,^spine,&f2>).

Tree Pruning 3 is formally defined as follows:

"For any nodes X, Y such that \(X.\text{cat}=C\) and \(C\text{OL}\) and \(X.d=Y\) (where 'L' is the set of category values that allow for lexical filling):
(a) ERASURE of \(X\),
(b) RE-ATTACHMENT of \(Y\) higher up."

(ibid. p.57)

Informally, this amounts to having any non-branching categorial node that is not lexically filled erased and its dependent node re-attached higher up (see Figure 4.6). Categorial nodes are the nodes that allow for lexical filling. They are, among others, V, Adv, P, PaP, PrP, and Aff. Furthermore:

"For the moment, NPs are considered possible categorial nodes, and are thus allowed to take lexical fillers (e.g. pronouns). As the internal syntax of NPs is developed further this anomaly will have to be removed."

(ibid. p.57)

Thus, we implement the rule as follows:

%Rule: Tree-Pruning-3

```
-> SD: ... # (&\text{CC}\>1 # ( ... )2 # )3 #
-> COND: "((x2: (S ... )) & (x3: (NP ... )))
-> SC: #1 #2 #3 #4 => #1 #3<^spine,&f1>
```
This rule looks for a node with a label C and only one daughter
((C<>1 ( ...))). If this C-node is an NP, its daughter may not be an S
^((x2: (S...)) & {x3: (NP...)}); this condition ensures that NP[S]-
nodes will not be pruned. The C-node, consisting of a labelled opening bracket (#2)
and a closing bracket (#4), gets erased. The features of the remaining node are
replaced by the features of the erased node (#3<^spine,&f1>). We give the fol-
lowing definition of the label variable C:

C = "V" | "N" | "NP" | "ADJ" | "ADV" | "P" | "PAP" | "PRP" |
+ "AFF" | "FV"

The fourth type of Tree Pruning as assumed by Seuren is trivial. He formally defines
it as: "For any node X that lacks a value for either fill (lexical filler) or d (depen-
dent(s)), erase X" (Seuren 1996, p.57). This simply implies the erasure of any node
that has neither a lexical filler nor any dependants (see Figure 4.7).

The implementation of this rule would also be trivial:

%Rule: Tree-Pruning-4
--> SD: ... # ( ) #
--> COND: None
--> SC: #1 #2 ==> #1

So far, however, we have not encountered any situation in which Tree Pruning 4 ap-
plies. The relevant configuration would only arise when all daughters of a node are
detached by some rule or rules, and this can never happen in the present set of im-
plemented rules. Thus there is no need to include Tree Pruning 4 in the
implementation.
4.3.4 Surface Category Assignment

Another procedurally induced routine is Surface Category Assignment. This routine changes the category of predicates into their lexically listed surface categories.

"SCA is obligatory whenever an SA-predicate is listed in the lexicon as belonging to a surface category different from V. It occurs typically as a corollary of any cyclic raising or lowering of the SA-predicate in question. When this raising or lowering involves a single lexical constituent v[X], SCA, if required, changes the category value 'V' into the surface category specified in the lexicon for X."  
(Seuren 1996, p.56)

This can be implemented as follows:

```plaintext
%Rule: Surface-Category-Assignment
--> SD: (S ... # (V<SCA>)1 ==2) #
--> COND: {(?&LEX: x2); (V<surf<>3>) &
         (x2 = (x2 node_category_change f3)) &
         x2 plus "<&f1,^SCA,^type,^rulf>"
--> SC: #1 #2 ==> #1 &x2
```

This rule looks for a lexically filled predicate marked for Surface Category Assignment ((V<SCA> ==2)). The filler is looked up in the lexicon and the enlisted surface category is retrieved {(?&LEX: x2); (V<surf<>3>)}. This surface category replaces the V-label (#1 &x2 with x2 = (x2 node_category_change f3)) and x2 plus "<&f1,^SCA,^type,^rulf>"

Yet, the rule is a little more complex than that:

"If, however, the raising or lowering involves a two-layered V cluster SCA assigns phrase status to the copy node dominating the cluster, besides assigning the correct surface category to the original V. No provision seems needed for the application of SCA to V-clusters with more than two layers."
(Seuren 1996, p.56)

Hence we extend the implementation:

```plaintext
--> SD: (S ... # (V<SCA>)1 ...2) #
--> COND: {((x2: (V ==3))) & {(?&LEX: x3); (V<surf<>4>) |
      ((x2: (V<SCA>)4> ( ... )5 ( ... )6 )) &
      x5 plus "<SCA>" & (x3 = '(x5 x6'))} &
      (x3 = (x3 node_category_change f4)) &
      x3 plus "<&f1,^SCA,^type,^rulf>"
--> SC: #1 #2 ==> #1 &x3
```

If the node marked for Surface Category Assignment is a branching node rather than a lexically filled one, then the applicable surface category is given as a feature value on the node itself instead of in the lexicon (x2: (V<SCA>)4> ( ... )5 ( ... )6 ). Note that this feature value is added onto the node by the rule that created it, i.e. by Object Incorporation (see section 4.3.8). Besides assigning the correct surface category to the branching node (#1 &x3 with x3=(x3 node_category_change f4)), the rule also assigns the triggering feature for Surface Category Assignment to its left daughter (x5 plus
"<SCA>". Since the rule is applied exhaustively, the daughter node will receive its own surface category in the next application of the rule.

If no surface category is given as a feature value on the branching node, then the default surface category V should be used:

\[
\text{COND: } ((\{x2: \text{(V === 3)}\} \& \{\text{?&LEX: x3: (V<surf<4>)}\} \mid \{(\{x2: \text{(V<SCA><4>) (~...5 (~...6 )}\} \& x5 \text{ plus } "<SCA>" \& (x3 = '\text{(x5 x6)'})) \mid ((\{x2: \text{(V ...3)}\} \& (f4 = "<V>"))) \& (x3 = (x3 node_category_change f4)) \& x3 \text{ plus } "<f1, ^SCA, ^type, ^rulf>"
\]

Surface Category Assignment has the following (in this case, rather minimal) effect on our example structure:

\[
\begin{array}{cccc}
S & \text{Surface-Category-Assignment} & --& S \\
\hline
V & NP & NP & V & NP & NP \\
\hline
V<type<PERF>, SCA> & AFF & V & V & AFF & V \\
\hline
\text{have} & \text{EN kiss John Mary} & \text{have} & \text{EN kiss John Mary} \\
\end{array}
\]

\subsection{4.3.5 Subject Raising}

The lexically induced rule Subject Raising is formally defined as follows:

"For any node $S_c$—d$=V\alpha-S^a_\beta$ or $S_c$—d$=V\alpha-S''_\beta$ where V.rulf$=<$SR$>$ ($\alpha, \beta$ are possibly null strings of nodes):

(a) DETACH the node N—cat—NP, N—p=S'd or Sod and N—fe—[SU],

(b) ATTACH N to the immediate left of /S'_d or /S''_d,

(c) redefine AF features for nodes under $S_c$.

(Seuren 1996, p.63)

Informally, this means that the subject NP of the lower V is placed in the position of its own S' or S''. The S is moved one position to the right.

This can be implemented as follows:

\[
\begin{align*}
\text{%Rule: Subject-Raising} \\
\text{--& SD: } (S \# (V<rulf<SR>) \ldots) & \text{ <= } # (S<^\text{type}<\text{FIN}), af>1) \\
(V \ldots) & \text{ <= } # (NP<af<SU>) \ldots) \# \\
\text{--& COND: None} \\
\text{--& SC: } #1 \#2 \#3 \#4 \\
\text{===> } #1 \#2<^\text{rulf}<\text{SR}> \#4<af>!&f1> \#3<af>
\end{align*}
\]

This rule selects the subject NP ((NP<af<SU>) \ldots) of the argument S' or S'' ($S<^\text{type}<\text{FIN}), af>1$) of the inducing predicate ((V<rulf<SR>) \ldots). This NP (#4) is placed in the position of the argument S (#3). Furthermore, the inducing rule feature on the predicate is removed (#2<^\text{rulf}<\text{SR}>). And the raised
NP takes over the argument function feature of the S-node (#4<af<!&fl>> #3<^af>).

If the argument-S contains no NP-subject, the rule blocks and the generation process is aborted. We therefore extend the implemented rule:  

\[ \rightarrow SD: (S \# (V<rulf<SR>> ...) \iff (S<type<FIN>,af>l> (V ...) (NP<af<I0>> ...)? # (NP<af<SU>> ...)?2 # \]

\[ \rightarrow COND: x2 \text{ is-empty } \rightarrow \text{ filter "No subject-NP to be raised"} \]

Application of Subject Raising also induces a subroutine called Downgrading 1. Seuren formalizes this routine as follows:

"For any node X such that X.cat=S and X.d=V-α (α may be $\Theta$; $\alpha$ does not contain a subject-node). NODE CATEGORY CHANGE: X.cat=S → X.cat=/S."

(ibid. p.58)

This means that an S is automatically demoted to the states of Verb Phrase (/S) if it is stripped of its subject constituent. Furthermore: "The superscripts in S', S', S'S and S" are maintained until the end of the Cycle but not for /S-nodes" (ibid. p.97).

We integrate this into the implemented rule:

\[ \rightarrow SC: #1 #2 #3 #4 \]

\[ \iff #1 #2<rulf<SR>> #4<af<!&fl>> #3<SLASH,^af,^type> \]

The rule applies as follows in the next cycle of our example structure:

\[ S \]

\[ V<rulf<SR>> S<type<PERF>,af<SU>> \]

\[ V \]

\[ NP<af<SU>> NP \]

\[ V \]

\[ AFF V \]

\[ seem have EN kiss John Mary \]

\[ Subject-Raising \iff S \]

\[ V \]

\[ NP<af<SU>> S<SLASH> \]

\[ V \]

\[ V \]

\[ AFF V \]

\[ seem John have EN kiss Mary \]

---

46 Another restriction on Subject Raising is given by Seuren (1996, p.99): "SR is sometimes subscripted subj or obj, indicating that SR applies only in subject or object position, respectively. When there is no subscript SR applies without restriction." This has not been accounted for in the implementation yet.
4.3.6 Particle-Insertion

A structurally induced rule we call Particle-Insertion (instead of Seuren’s language-
specific name To-Insertion) is responsible for inserting the particle to before infinit-
ives. Seuren describes the rule as follows:

"To-INSERTION is a corollary at the end of the cycle of /S-constituents not containing AUX,
unless the V-constituent under the parent of /S is marked [-to]. Select the immediate parent-
node P of the highest (leftmost) filled V-node in the V-cluster. P left-adopts the Particle node
Prt[to]."

(Seuren 1996, p.95)

The basic implementation is as follows:

%Rule: Particle-Insertion

\[\iffalse\]
\begin{verbatim}
SD: (S <=<
    (S<SLASH> <=< # (V ===) ==>) #
\rightarrow COND: (x1 = "(Prt-to)")
\rightarrow SC: #1 #2
    => #1 (&x1 >> #2<spine>)<^spine>
\end{verbatim}
\[\fi\]

This rule looks for the parent-node of the highest filled V-node (( (V ===) ==>) within the /S-constituent (S<SLASH>). This node left-adopts the particle node to (&x1 >> #2 with x1="(Prt-to)"). As usual with adoption by a V-
node, the spine features are updated.

Seuren’s description mentions two extra conditions: the /S-constituent must not
contain AUX, and the V-constituent under the parent of /S must not be marked [-to].
The first condition ensures that we are dealing with an infinitive. The second condi-
tion reflects the well-known fact that, although the default in English is for the
infinitive to be preceded by to, a small number of matrix verbs such as let and make
and the verbs sentiendi see and hear have an embedded infinitive without the ver-
ble, see for example (47a,b). These verbs are marked in the lexicon as [-to]-verbs.

(47)  a  I saw John kiss Mary.
      b  * I saw John to kiss Mary.

Thus, we extend the implementation:

\[\iffalse\]
\begin{verbatim}
SD: (S (V ===1) <=<
    (S<SLASH> <=< # (V (V ===) ==>) # ==>)2
\rightarrow COND: ^{([^?[LEX]: x1}: (V<-to>) &
    ^{(x2: ... (AFF "PRES")) | \{x2: ... (AFF "PAST"))} &
    (x3 = "(Prt-to)")
\rightarrow SC: #1 #2
    => #1 (&x3 >> #2<spine>)<^spine>
\end{verbatim}
\[\fi\]

Now, the /S-constituent must not contain a finite tense affix
(^{(x2: ... (AFF "PRES")) | \{x2: ... (AFF "PAST"))}, which
Transformational grammar for English

is equivalent to saying that it must not contain AUX.\footnote{The rule AUX-Delimitation (see section 4.4.1) adds features to mark the AUX-area. Since we implement AUX-Delimitation as a Postcyclic rule, however, the AUX-features are not available at this stage.} Also, it is checked in the lexicon that the V-constituent under the parent of /S ((S (V ===1)) is not marked [-to] (\langle\{?[\text{LEX}]: x1\}: (V<-to>)).

Sometimes, the V-node that is to adopt to may be the only V present under /S, i.e. the V-“cluster” may consist of only one node. In that case, obviously, it is not the parent-node P that left-adopts the particle, but rather the V-node itself:

\[
\rightarrow SD: (S (V ===1)) ==
\]

\[
(S<\text{SLASH}> == (\{V \}? (V ===) \{==>()? \# ===>)2
\]

Seuren notes that with some verbs, such as help, the particle to on the embedded infinitive is optional. See, for example, sentences (48a.b).\footnote{There are also verbs which have the particle to only in the passive, not in the active (as in John was made to kiss Mary versus I made John kiss Mary). This has not been implemented yet.}

(48) a John helped Mary get up.

b John helped Mary to get up.

This is accounted for in the implementation by means of an alternative SC:

\[
\rightarrow SC: \#1 \#2
\]

\[
== \#1 \{x3 >> \#2<\text{spine}>\}<\text{spine}>
\]

\[
== \{(?[\text{LEX}]: x1): (V<to>)\} \#1 \#2
\]

The effect of Particle-Insertion on our example structure is:

In the next cycle, our example structure only undergoes rules that I have already discussed: one lexically induced rule (Lowering-to-V) and two corollaries (Tree-Pruning-1 and Surface Category Assignment), respectively:
And in the topmost cycle, our example structure again only undergoes rules that have already been discussed. This time, the predicate induces two rules (Subject-Raising and Lowering-to-V), followed by one corollary (Surface Category Assignment):
This structure is the so-called Shallow Structure and will be input to the Postcyclic rules, which are discussed in section 4.4.

First, I will turn to the remaining Cyclic rules that do not apply in our example structure. There are two more lexically induced rules (Subject Deletion and Object Incorporation), two structurally induced rules (Extraposition and Copula Insertion), and one more corollary (Complementizer-Insertion).

4.3.7 Subject Deletion

The rule Subject Deletion is used in so-called control structures, to remove the subject of a complement clause if it refers to the same entity as the subject or indirect object of the higher clause. This is illustrated in examples (49a,b), where the subject and the indirect object of the matrix verb want, respectively, refer to the same entity as the implicit (i.e. deleted) subject of the embedded verb kiss.

(49) a John wanted to kiss Mary.
    b John wanted me to kiss Mary.
Seuren gives the following formal definition of the deletion rule:

“For any node $S_{a} \rightarrow d = V_{a} \alpha \sigma_{a} d_{a} \beta$ or $S_{a} \rightarrow d = V_{a} \alpha \text{-NP} \sigma_{d} / \text{NP} \sigma_{d} \beta$ where $V_{a} \text{rulf} = <SD>$ (where $\alpha$, $\beta$ are possibly null strings of nodes), if $\alpha$ or $\beta$ contains NP (x) or $S_{a} \rightarrow d = V_{a} \text{-NP} \gamma$ and $\sigma_{d} / \sigma_{d} \beta$ where $\alpha \text{rulf} = <SD>$ (where $\gamma$, $\delta$ may be 0), (a) DETACH the node $\sigma_{d} \beta [x]$, (b) DELETE the detached constituent.” (Seuren 1996, p.71)

Or, less formal and easier to read:

“SD (only with object $S^2$/S^2” or NP[S^2/S^2]): Select the NP to the right of lower V marked [SU]. If this NP is of the form NP[x] and is controlled by an NP-argument (a) of the inducing V itself (vertical SD) or (b) of its subject-S (horizontal SD): delete NP[x]. Horizontal SD is limited to prepositional predicates followed by a, possibly tensed, gerund.” (ibid. p.95)

Vertical Subject Deletion can be implemented as follows:

%Rule: Subject-Deletion

$\rightarrow$ SD: (S # (V$<$rulf$<$SD$>$) . . . ) (NP . . . ) # (NP$<$<<$>$$>$ . . . ) {(NP) ? # (S$<$^type$<$FIN$>$) (V . . . ) # (NP$<$af$<$SU$>$) "$x" $>$ #

$\rightarrow$ COND: None

$\rightarrow$ SC: #1 #2 #3 #4 #5

$\Rightarrow$ #1 #2$<$rulf$<$SD$>$ #3$<$<<$>$ #4

This rule looks for a predicate with the relevant rule feature (V$<$rulf$<$SD$>$)) which has as its first or second argument an NP that can act as a controller (NP$<$X$>$). If the next argument is a non-finite complement clause (S$<$^FIN$>$) with an NP-subject that refers to the same entity as the controller ((NP$<$af$<$SU$>$) "$x" $>$ ), then this latter NP (#5) is deleted. The rule feature and the referential feature are also deleted (#2$<$rulf$<$SD$>$ #3$<$<<$>$).

To include horizontal Subject Deletion, we could extend the SD:

$\rightarrow$ SD: (S # (V$<$rulf$<$SD$>$) . . . ) (NP . . . ) {(S (V . . . )? # (NP$<$X$>$ . . . ) (==) )? {(NP) ? # (S$<$^type$<$FIN$>$) (V . . . ) # (NP$<$af$<$SU$>$) "$x" $>$ #

Now, the controller itself may be embedded in a subject clause (((S (V . . . ))? (NP$<$X$>$ . . . ) (==) )?). However, as Seuren remarks, horizontal Subject Deletion is limited to prepositional predicates followed by a gerund. Since such constructions are not accounted for otherwise by Seuren’s grammar (i.e. they are not included in the set of sentences as given on the CD-ROM), there is no need to extend the rule of Subject Deletion in this respect.

If there is no referential identity between the higher and the lower subject, Subject Deletion does not apply. In English, most of the Subject Deletion predicates are

---

40 As such, the rule could just as well be treated as a structurally induced rule instead of a lexically induced one, since the occurrence of the two necessary coreferential NPs is already lexically determined. However, treating Subject Deletion as a structurally induced rule would lead to an ordering paradox: on the one hand it must precede the lexically induced rule of Subject Raising, but on the other hand all lexically induced rules precede all structurally induced rules (see p.90-93). Further study would be needed to find a solution to this problem. For now, we keep to Seuren’s (1996) formalization and leave Subject Deletion to be lexically induced.
marked for Subject Raising to apply instead in those cases. Otherwise, the sentence cannot be processed. This can be integrated into the implemented rule as follows:

```
--> SD: (S # (V<rulf<SD>l> ...) (NP ...) # (NP<X> ...) } (NPX)? #
(S<type<FIN>> (V ...) # (NP<af<SU>> ...)) #

--> COND: (^{x2: "x"}) & (^{f1 features "<SR>"]) ->
filter "No referential identity"

--> SC: #1 #2 #3 #4 #5

==>[^[x2: "x"]] #1 #2<rulf<SD>> #3^<X> #4

==>[^[x2: "x"]] #1 #2<rulf<SD>> #3^<X> #4 #5
```

Note that the rule now accounts for three different situations:

- If the lower subject satisfies the anaphoric reference condition ({x2: "x"}) it is deleted.
- If the anaphoric reference condition fails (^{x2: "x"}) and there is no alternative rule marking for Subject Raising (^{l features "<SR>"}), the tree is filtered out(filter "No referential identity").
- If the anaphoric reference condition fails but the inducing predicate is also marked for Subject Raising, then the structure is unaltered except for the removal of the Subject Deletion feature and the anaphoric reference feature (#2<rulf<SD>> #3^<X>).

Application of Subject Deletion induces the subroutine Downgrading 1, just as Subject Raising does (see p. 104-106). In the same way as was done for Subject Raising, we incorporate the subroutine into the rule of Subject Deletion:

```
--> SC: #1 #2 #3 #4 #5

==>[^[x2: "x"]] #1 #2<rulf<SD>> #3^<X> #4<SLASH,^af,^type>

==>[^[x2: "x"]] #1 #2<rulf<SD>> #3^<X> #4 #5
```

The rule has the following effect on the relevant local tree for example sentence (49a):

```
S
V<rulf<SD,SR>> NP<X,af<SU>> S<type<PERF>,af<DO>>
|    |    |
|    |    |
want   John
  V V
  V V
SIM kiss x Mary

Subject-Deletion =>

S
V<rulf<SR>> NP<af<SU>> S<SLASH>
|    |    |
|    |    |
want   John
  V V
  V V
SIM kiss Mary
```
4.3.8 Object Incorporation

One final lexically induced rule that has not yet been discussed is Object Incorporation.\(^{50}\) This rule is responsible for the incorporation of NP-objects by prepositional predicates (and a few other predicate types such as quantifiers).\(^{51}\) Formally, the rule reads:

“For any node \(N_i\): if \(\text{cat}=[\text{NP or S}]\) \(d=\text{V} [\text{NP or S}]\rangle \langle \text{NP}\rangle \text{ or } \alpha \text{ is null): (a) detach the node } N_i, \text{ (b) } \text{V } \alpha \text{-adopts } N_i.\) (Seuren 1996, p.72)

This can be implemented straightforwardly:

```
%Rule: Object-Incorporation
--> SD: (S # (V<rulf<OI>) ==) #
    (#ARG ... ) (NP ...) # #
--> SC: #1 #2 #3 #4
    ===> #1 ( #2<rulf> << #4<af> ) #3
```

Any remaining rule features need to be passed on to the new mother node:

```
--> SD: (S # (V<rulf<OI>1) ==) #
    (#ARG ... ) (NP ...) # #
--> SC: #1 #2 #3 #4
    ===> #1 ( #2<rulf> << #4<af> )<rulf<&f1,^OI>> #3
```

The surface category of the new mother node needs to be determined at this stage, so it can be assigned by Surface Category Assignment later:

```
--> SD: (S # (V<rulf<OI>1) ==) #
    (#ARG ... ) (NP ...) # #
--> COND: {?[@LEX]: x2}: (Vsurf<P>>& ) --> (f3 = "<surf<PP>>")
-->
```  #1 #2 #3 #4
    ===> #1 ( #2<rulf> << #4<af> )<rulf<&f1,^OI>,&f3> #3
```

Let me illustrate this rule for the prepositional by-phrase in the passive counterpart (50) of our example sentence.

(50) Mary seemed to have been kissed by John.

Application of Object-Incorporation has the following effect on the relevant cycle of the tree structure of this sentence:

\(^{50}\) Just like Subject Raising. Object Incorporation too could be treated as a structurally induced rule instead (see footnote 49 on p.110). Again, to avoid the ordering paradox, we follow Seuren’s formalization in regarding it as lexically induced however.

\(^{51}\) Object Incorporation is also a productive rule within the lexicon where it is responsible, for example, for fixed combinations like take care of and pay attention to in English and compound verbs like stofzuigen (vacuum cleaning, literally “dust-sucking”) in Dutch (see Seuren 1996, p.71-72).
Note that the rule feature (rulf<\text{L}<\text{R}>>) on the newly created V-node will ensure that the whole preposition phrase is lowered to the far right (see section 4.3.2).

### 4.3.9 Extraposition

Seuren assumes a structurally induced rule IT or Extraposition to insert a dummy subject \(it\) with finite S-complements.\(^\text{52}\) The formal definition is:

"For any node \(S'_{\text{e}} - d = \text{V} - \gamma - \alpha\) (\(\gamma\) is \(S_d - d = \text{NP} - /\text{S} - \beta\) or \(\text{NP} - d = \text{NP} - /\text{S} - \beta\); \(\alpha\) and \(\beta\) may be \(\emptyset\)): (a) create \(N - \text{cat} - \text{NP}\), fill "it", (b) insert \(N\) into \(S'_{\text{e}}\) to the immediate right of \(\text{V}\)."

(Seuren 1996, p.59)

This rule is responsible for the insertion of the dummy subject \(it\) in sentences like (51a). Note that this insertion of dummy \(it\) is obligatory, since without it, the ungrammatical (51b) would be generated.

\[(51)\]
\[a\] It seems that John has kissed Mary.
\[b\] * That John has kissed Mary seems.

The rule is basically implemented as follows:

\(^\text{52}\) We prefer to use the less language-specific name Extraposition for this rule.
%Rule: Extraposition

\[ \rightarrow SD: (S (V ...) \# \{(NP \})? (S\text{type}<>FIN\text{>} \#\ldots) \]

\[ \rightarrow COND: \text{None} \]

\[ \rightarrow SC: \#1 \]

\[ \Rightarrow \#1 (NP "it") \]

If the tree structure contains an embedded finite clause \((S\text{type}<>FIN\text{>} \ldots)\), possibly headed by an NP-node \(\{(NP \}\?)\), a dummy subject is inserted to the immediate right of the V-node \(#1 (NP "it")\).

With embedded clauses headed by an NP, the situation is actually somewhat more complicated. Compare sentences (52a,b) with (51a,b) above.

\( \text{(52)} \)

- a It is likely that John has kissed Mary.
- b That John has kissed Mary is likely.

Seuren accounts for the difference in grammaticality of the b-cases by means of a different status of the embedded clause: \textit{seem} selects a bare S” as its subject argument, whereas \textit{likely} selects an \(s_{NP}[S"]\). Furthermore:

"With a subject of the form \(s_{NP}[S"]\) the rule is not obligatory but preferred. It seems that this preference can be overridden under conditions to do with the topic-comment modulation of sentences. Unfortunately, these conditions are not entirely clear. (...) Here we must be content with an incomplete formulation of the application conditions of the rule. We say that when a subject-clause \(s_{NP}[S"]\) functions as topic \textit{IT} is obligatory. But when a subject-clause \(s_{NP}[S"]\) functions as comment \textit{IT} is probably best regarded as optional.”

(\textit{ibid.} p.59)

Although topic and comment functions are discourse-related notions that are normally not expressed within the tree structure, we can implement the relation between the topic-comment modulation of a sentence and the optionality of Extraposition in the following, provisional way:

\[ \rightarrow SD: (S (V ...) \# \{(NP\text{<^comment>}})?)\#1 (S\text{type}<>FIN\text{>} \#\ldots) \#\]

\[ \rightarrow SC: \#1 \#2 \]

\[ \Rightarrow \#1 (NP "it") \#2 \]

\[ \Rightarrow [(x1 \text{is_empty})] \#1 \#2\text{<comment>} \]

Now, if the embedded clause is headed by an NP-node \(\{(NP)\#1\text{ and} \nearrow(x1 \text{is_empty})\), the insertion of the dummy optionally remains undone. In that case, the embedded clause receives a feature that marks it as comment \(#1 \#2\text{<comment>}\).

Seuren (1996, p.58) also notes that the rule has the additional effect of moving the original sentential subject to object position if it is not already there (it is this effect which is reflected in the name Extraposition). This is why we find (53a) rather than (53b).

\( \text{(53)} \)

- a It seems to me that John has kissed Mary.
- b * It seems that John has kissed Mary to me.

This is integrated into the implemented rule as follows:
Now, the embedded clause is moved across its right-hand sister node(s) (#3 #2), if this node (or one of these nodes) is not the direct object itself (^{x3: \Longrightarrow \langle af<DO>> ...}).

Let me illustrate this rule by means of the finite-complement counterpart (51a) of our example sentence. Application of Extrapolation has the following effect on the relevant cycle of the tree structure of this sentence:

4.3.10 Copula Insertion

The copula verb is not base-generated by the formation rules, but inserted in the Cycle by the transformational rules. The structurally induced rule Copula Insertion is defined as follows: "S' directly above V_{Adj}, Adj or V_{NP} α-adopts V_{cop}[be]" (Seuren
1996, p.95). In other words, if the nuclear predicate is an adjective or a noun phrase rather than a verb, a copula is inserted. Next, the copula induces the cyclic rule of Predicate Raising, which raises the nuclear predicate to be adopted by the copula and causes the intermediate S to disappear through Downgrading 2 (see section 5.2.1). The two-step process is illustrated in Figure 4.8, which shows the relevant cycle of the copular sentence in (54).

\[\text{Figure 4.8 The two-step process of Copula Insertion and Predicate Raising according to Seuren (1996).} \]

(54) John was ill.

A possible implementation of Copula Insertion would be:

```
%Rule: Copula-Insertion
--> SD: (S<type<LEX>> (V ===1) ==)) #
--> COND: {(?&LEX: x1): (V<surf<ADJ|NP>>) &
(x2 = "(V<rulf<BE>>-be)")}
--> SC: #1 ==) &x2 >> #1
```

This rule looks for an adjectival or nominal predicate ((V ===1) with ((?&LEX: x1): (V<surf<ADJ|NP>>) with an S°-mother (S<type<LEX>>)). The S°-mother left-adopts a copula with the rule feature that is to trigger Predicate Raising (&x2 >> #1 with (x2 = "(V<rulf<PR>>-be)").

However, a technical problem arises with this implementation when the adoption takes place. Since the active node is a cyclic S-node, the copy node is also a cyclic node. But which of the two now constitutes the cycle with which the system has to continue? More generally put, generating a new cyclic node on top of the active cyclic node is a violation of the principle of strict cyclicity, since it affects the tree structure outside the present domain.

A solution to this problem readily presents itself: why not have the adjectival or nominal predicate directly adopt the copula, instead of first having the S-node adopt it and then having the predicate raised by a separate rule? In other words, Seuren’s two-step process can easily be replaced by a one-step process, as illustrated in Figure 4.9. This solution has the additional advantage of rendering the rule of Predi-
cate Raising superfluous in the rule system for English, because this rule is not induced by any other predicate.

```
/   \       /   \      /   \      /   \      /   \      /   \      /   \      /   \      /   \      /   \
| Cl  |      | S⁰  |      | S⁰  |      | V    |      | NP   |      | V    |      | NP   |      | VCop |      | ADJ  |
|____|      |____|      |____|      |____|      |____|      |____|      |____|      |____|      |____|      |
| ill |      | V   |      | ill |      | John|      | John|      | be   |      | ADJ  |      |       |
```

Figure 4.9 A one-step process of Copula Insertion.

Thus, we rather implement the rule of Copula Insertion as follows:

```
---> SD: \{S<type<LEX>> # (V =#1) \#
---> COND: \{(?&LEX: x1): (V<surf<ADJ|NP>>) \&
           (x2 = "(V-be)")\}
---> SC: #1 #2 =>> #1 \{ &x2 >> #2<spine,SCA> \}<type,^spine,^SCA>
```

Now, the copula is adopted by the adjectival or nominal predicate itself, which additionally receives a feature to trigger Surface Category Assignment (&x2 >> #2<SCA>).

Note that this line of reasoning is another instance of the merits of our technolinguistic approach. A technical problem of the implementation turns out to be a conceptual problem in the formalization. It leads us to adapt both of them while leaving the underlying theoretical principles intact (the main principle being the conviction that a copula is a purely syntactic item which does not contribute to the semantics of a sentence and hence does not belong in the SA).

Seuren mentions two extra conditions on the application of the rule: ‘In general the main condition is that the parent node of V_{adj} or V_{NP} be, at that stage in the Cycle, a full S (not /S), and, moreover, the V_{adj} or V_{NP} in question must, after application of all lexically induced cyclic rules, still be in the original position’ (1996, p.62). As far as I can see, these conditions are always met automatically in the languages under consideration, so there is no need to express them explicitly in the implementation.

Taking the first cycle of sentence (54), for example, the rule applies as follows:
4.3.11 Complementizer Insertion

Finally, a corollary is responsible for the insertion of a complementizer in embedded finite declarative clauses.\textsuperscript{53} Seuren calls this rule *Thar*-insertion, but since it applies not only in English we prefer the more general name Complementizer Insertion. Seuren defines the rule as: "That-INSERTION is a corollary at the end of the Cycle of S" or \textit{np}[S"] constituents occurring as argument term to a lexical predicate: S" left-adopts \textit{comp}[that]" (1996, p.96). The basic implementation of this rule is quite simple:

%Rule: Complementizer-Insertion
\begin{align*}
\text{sd}: & (S<\text{type}<\text{LEX}>> \iff \{(\text{NP })? \# (S<\text{type}<\text{FIN}>> \ldots ) \# \}
\text{cond}: & (x_1 = "(\text{COMP-that})")
\text{sc}: & #1 \#2 \implies \#1 \& x_1 \implies \#2
\end{align*}

This rule looks for a finite S ((S<type<FIN>> ...)), optionally headed by an NP-node \((\{(\text{NP })?\}\), and embedded in a lexical S (S<type<LEX>>). The finite S left-adopts the complementizer (\&x_1 \implies \#2 with \(x_1 = "(\text{COMP}-\text{that})\)). Note that in our implementation, the rule actually operates at a different stage than is expressed in Seuren’s definition, namely in the next higher cycle. The rule could not be implemented with the finite S itself as the cyclic node, because then we would not be able to express that the finite S must be embedded rather than a regular main S".\textsuperscript{54}

We must ensure that the rule only inserts one complementizer per clause. Hence we add:

\begin{align*}
\text{sd}: & (S<\text{type}<\text{LEX}>> \iff \{(\text{NP })? \# (S<\text{type}<\text{FIN}>,^\text{THAT}>> \ldots ) \# \}
\text{cond}: & (x_1 \implies ^\text{THAT}>> \#2)
\text{sc}: & #1 \#2 \implies ^\text{THAT}>> #1 \{ \& x_1 \implies \#2 \}
\end{align*}

Now, the rule adds a feature to the newly created S-node (\((\& x_1 \implies \#2 <\text{THAT})\)) to indicate that it already has a complementizer and thus prevent the rule from applying again in another cycle (S<type<FIN>>, ^\text{THAT}).

Additionally, the usual features must be added because adoption takes place:

\textsuperscript{53} For now, we leave embedded questions out of account.
\textsuperscript{54} More precisely, it \textit{would} in fact be possible to single out only embedded S'-nodes by means of the \textit{<unit>-feature}, but it is not possible (without introducing an ad-hoc feature) to single out the top node of such embedded clauses. This cycle problem may also be taken to indicate that the rule should perhaps be implemented as a post-cyclic rule instead, but we have not investigated this possibility any further.
Let me illustrate this rule again by means of the finite-complement counterpart of our example sentence, which has been used before to illustrate the application of Extrapolation (see sentence (51) on p.113). In fact, Complementizer Insertion takes place immediately after Extrapolation, on the same cycle:

\[
S \text{ type} \langle \text{LEX} \rangle \to
\]

\[
V \quad \text{NP} \quad S \text{ type} \langle \text{FIN} \rangle, \text{af} \langle \text{DO} \rangle, \text{unit} \langle 1 \rangle, \text{spine} \]

\[
\quad \text{NP} \quad S \langle \text{SLASH} \rangle
\]

\[
\quad \quad \text{AFF} \quad V
\]

\[
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \.;
%Grammar: Postcycle
---> Rule: AUX-Delimitation
---> Grammar: SIM-Deletion
---> Grammar: Adverb-Placement
---> Grammar: Attraction
---> Grammar: Do-Support
---> Grammar: Affix-Handling
---> Grammar: Mince
---> Stop

These rules and subgrammars do not apply cyclically but to the whole tree structure (that is, they deal with root phenomena rather than local phenomena), in the order in which they are called. I will also treat them in that order here.

4.4.1 AUX-Delimitation

An important notion within Semantic Syntax is that of the AUX-area: the part of the tree that will end up as the finite verb. However, this part is not a constituent of the tree but an area, a construct that is formally defined as follows:

"An area is a constituent C minus a subconstituent D of C. Any constituent or subconstituent thus forms an area (when D=Ø), but we normally speak of an area only when D≠Ø. An area has an entrance, which is the parent connection of N₄ dominating C, and an exit, formed by the parent connection of N₄ dominating D. If D=Ø the node N₄, such that N₄p=N₄ in cases where D≠Ø, adopts a temporary node EXIT (to the right if C is right-branching; to the left if C is left-branching)."

(Seuren 1996, p.47-8)

The AUX-area is defined at Shallow Structure level and various postcyclic rules are sensitive to it, such as Adverb Placement and Question Formation. The area will be turned into a constituent by the postcyclic rule Affix Handling. Seuren's definition of the AUX-area is as follows:

"Throughout the Postcycle, AUX consists of that part of the V-cluster dominated by the V-node immediately dominating Aref[PRES/PAST], down to and including the first lexically filled V-node (Ø counts as a lexical filler). Cut through the sister branch of this V, if any."

(Seuren 1999b, p.6)

The implementation closely mirrors this definition:

%Rule: AUX-Delimitation
---> SD: (S ... (V # (AFF<^AUX> {"PRES"|"PAST"}) ... # (V ==)) #
---> COND: None
---> SC: #1 #2 #3 ==> #1 #2<AUX> #3<AUX>

---

55 This definition is taken from Seuren 1999b (an unpublished manuscript) instead of Seuren 1996. This new definition differs slightly from the original one due to the changed view on level 2 adverbials that we have developed above (see section 3.2.5).
This rule looks for the finite affix (\(\text{AFF}^{\wedge}\text{AUX} > \{"\text{PRES}" | "\text{PAST}"\}\)) and the first lexically filled V-node to the right (\((V == )\)). These two nodes each receive a feature that marks them as being part of the AUX-area (\(#2<\text{AUX}> #3<\text{AUX}>\)).

Seuren mentions two special situations:

"NB1: AUX is redefined according to the above procedure after treatment of \(\emptyset\) (\(\emptyset\)-be-DELETION and Do-SUPPORT), and, for languages with V-cluster clitics (e.g. French), after CLITIC MOVEMENT. AUX is then fixed and is an island: no material may be moved into or out of AUX, but internal changes are allowed.

NB2: In English, AUX may incorporate \textit{not} after ADVERB PLACEMENT." (Seuren 1996, p.95)

In the implementation, these situations are not catered for by re-calling the rule AUX-Delimitation, but rather within the relevant rules themselves. Thus, the rules SIM-Deletion, Do-Support and Adverb Placement each add or remove AUX-features as necessary.

The rule has the following effect on the example structure:

\[
\begin{align*}
S & \\
\text{NP} & \quad S<\text{SLASH}> \\
& \quad \text{V} \\
& \quad \text{AFF} \quad \text{V} \\
& \quad \text{V} \quad \text{V} \quad \text{Prt} \quad \text{V} \\
& \quad \text{V} \quad \text{V} \\
& \quad \text{AFF} \quad \text{V} \\
& \quad \text{John \ PAST \ SIM \ seem \ to \ have \ EN \ kiss \ Mary} \\
& \quad \text{AUX-Delimitation} \quad \Rightarrow
\end{align*}
\]

56 The AUX-area is implemented by means of a feature on the lexical nodes, leaving implicit that their mother nodes as well as any intervening non-V nodes also belong to the area. It might also be possible to use features to mark the entrance and exit of the area but we did not examine this possibility any further, although it would probably stay closer to the formal definition of the area notion. The main reason for not using the entrance and exit notions is that they are likely to lead to complex rule definitions when the AUX-area is cluster-final.

57 It remains to be seen whether re-calling the rule AUX-Delimitation would result in a (technologically) better implementation.
4.4.2 SIM-Deletion

The first real postcyclic rule is to remove the tense-operator SIM from the tree structure. Seuren (1996) calls this rule Ø-(be-)deletion, but since we use the abstract filler SIM (=simultaneous) instead of Ø for the perfective tense-operator (see p.29-32) we call it SIM-Deletion. In English, the operator is only removed under certain conditions. If not removed now, it will be either removed or replaced by the dummy verb do in the later rule Do Support.

In the implementation, the rule has its own subgrammar because it triggers the corollary Tree-Pruning (which we have already seen in the Cycle, see p.99-103):

```
%Grammar: SIM-Deletion
--> Rule: SIM-Deletion
--> Grammar: Tree-Pruning
--> Stop
```

For English, the rule is defined as follows: “If the right brother of √[Ø] equals or directly dominates √[be] delete √[Ø]” (Seuren 1996, p.96). This can be implemented as follows:

```
%Rule: SIM-Deletion
--> SD: (S ... (V # (V "SIM") # ((V )? # (V &BE) ) #
--> COND: None
--> SC: #1 #2 #3 #4 ==> #1 #3 #4
```

This rule looks for the simultaneous tense operator ((V "SIM")) and checks whether its right brother is be ((V &BE)) or dominates be ((V )?). If so, the tense operator (#2) is removed. Note that the variable &BE stands for the various instances of be distinguished in the lexicon, such as the copula be, the progressive auxiliary bel₁, and the passive auxiliary be₂.
SIM-Deletion is one of the rules that involve a re-definition of the AUX-area (see section 4.4.1).\footnote{This might be reason to switch the order of AUX-Delimitation and SIM-Deletion around; however, we have not investigated this any further yet (see also footnote 57, p.121).} We integrate this into the rule as follows:

\[
\begin{align*}
\rightarrow \text{SP:} & \; (S \ldots (V \# (V \leftarrow 1 "\text{SIM}") \# ((V) \# (V \& \text{BE}) \# \\
\rightarrow \text{SC:} & \; \# 1 \# 2 \# 3 \# 4 \implies \# 1 \# 3 \# 4 <\& f 1 >
\end{align*}
\]

Now, the feature of the removed tense operator (i.e. <AUX>) is copied onto the be-node (#4<& f 1 >).

SIM-Deletion does not apply in our example structure, since that structure does not contain a SIM-be-sequence. Let me illustrate the rule with the copular sentence (54), which we have already seen above (p.116) and which does contain such a sequence:

(54) John was ill.

4.4.3 Adverb Placement

The postcyclic rule of Adverb Placement only applies in English. It is responsible for removing any adverbial from the AUX-area. The rule comes in two mutually exclusive varieties, depending on whether or not the cluster contains the adverbs not or EMPH. The formal definitions are the following:\footnote{Again, this definition is taken from Seuren 1999b, because there it was adapted to the new view on level 2 adverbials (see also footnote 55 on p.120).}

For any Adverb area of the V-cluster inside AUX:

a. If AUX contains $A_\text{adj}[\text{not}]$ or $A_\text{adj}[\text{EMPH}]$: (1) The Adverb subarea above $A_\text{adj}[\text{not}]$ or $A_\text{adj}[\text{EMPH}]$ is moved to entrance of AUX. (2) The Adverb subarea down from and including $A_\text{adj}[\text{not}]$ or $A_\text{adj}[\text{EMPH}]$ is moved to the exit of AUX.

b. If AUX does not contain $A_\text{adj}[\text{not}]$ or $A_\text{adj}[\text{EMPH}]$: (1) If AUX contains $V[2]$ or is cluster-final, the whole Adverb area is moved up to entrance of AUX. (2) Otherwise the whole Adverb area is moved down to exit of AUX.

(Seuren 1999b, p.6)

To see the effects of the two varieties, refer to sentences (55a,b) and (56a,b). In (55a), Adverb Placement a-2 has applied to place the adverb not below the AUX-area (which surfaces as the auxiliary verb do; see section 4.4.5). In (55b), in addi-
tion, also Adverb Placement a-1 has applied to place the adverb *usually* above the AUX-area. In (56a), Adverb Placement b-1 has applied to place the adverb above the AUX-area (which contains the main verb *kiss*). In (56b), instead, Adverb Placement b-2 has applied to place the adverb *never* below the AUX-area (which contains the tense auxiliary *have*).

(55)  
  a. John does not kiss Mary.
  b. John usually does not kiss Mary.

(56)  
  a. John never kissed Mary.
  b. John has never kissed Mary.

The two varieties of Adverb Placement are implemented as two separate rules, which we group into a subgrammar:

```
%Grammar: Adverb-Placement
 --> Rule: Adverb-Placement-a
 --> Rule: Adverb-Placement-b
 --> Stop
```

Adverb Placement a (including both a-1 and a-2) is implemented as follows:

```
%Rule: Adverb-Placement-a
 --> SD: (S ... # (V (AFF<AUX> ==) # ... # (V (ADV ==1) ... #
        {(V )? (V<AUX> ==) #
 --> COND: {x1: "EMPH"} | {x1: "not"}
 --> SC: #1 #2 #3 #4 #5
         => #1 #3 #2 #5 #4
```

The rule looks for an adverb *EMPH* or *not* ((ADV ==1) with {x1: "EMPH"} | {x1: "not"}) within the AUX-area. Any AUX-internal nodes above this adverb (#3) are placed at the entrance of the area (a-1), whereas the adverb itself plus any AUX-internal nodes below it (#4) are placed at the exit (a-2).

Another aspect of this rule is the optional incorporation of *not* into the AUX-area. If Adverb Placement a applies, the adverb *not* may become part of the AUX-area leading to the common reduction of *not* to *n’t* as in (57), the colloquial counterpart of example (55a). Seuren (1996, p.96) formally defines this as: “Adv[not]. if at the exit of AUX, and not immediately followed by Adv[EMPH], may become part of AUX.”

(57)  
  John doesn’t kiss Mary.

This can be integrated into the rule by means of an additional Structural Change:

```
 --> SD: (S ... # (V (AFF<AUX> ==) # ... # (V (ADV ==1) ...2 #
        {(V )? (V<AUX> ==) #
 --> SC: #1 #2 #3 #4 #5
         => #1 #3 #2 #5 #4
        
    => [{x1: "not"} \& ^{x2: (V (ADV "EMPH")})]
           #1 #3 #2 #5 #4<AUX>
```
Now, if the triggering adverb is not (\{x1: "not"\}) and it is not immediately followed by EMPH (\{x2: (V (ADV "EMPH"))\}), a second tree arises in which the adverb not also receives the AUX-feature (#3<AUX>).

Let me illustrate the operation of this rule in the generation of the sentences (55a) and (57), which share their underlying structure:

The b-variety of Adverb Placement is basically implemented as follows:

%Rule: Adverb-Placement-b

\[\rightarrow SD: (S \ldots \# (V (AFF<AUX> \#\#) \# (V (ADV \#\#) \ldots \# (V (V<AUX> \#\#)) \#)
\]
\[\rightarrow COND: None
\]
\[\rightarrow SC: #1 #2 #3 #4
\]
\[\Rightarrow [\{x1: "SIM"\}] #1 #3 #2 #4
\]
\[\Rightarrow [\{x1: "SIM"\}] #1 #2 #4 #3
\]

This rule looks for any adverb in the AUX-area (\(V (ADV \#\#)\)) and places it, together with its mother-node and any other AUX-internal nodes, at the entrance of the area if AUX contains the simultaneous tense-operator (\(V (V<AUX> \#\#1)\) with \{x1: "SIM"\}), and at the exit of the area otherwise (\(^{\{x1: "SIM"\}}\)).

---

60 This implementation does not account for the situation where the adverb area should be moved to the entrance because AUX is cluster-final, as in John never sleeps.
4.4.4 Attraction

Recall that in Semantic Syntax topicalization and questions are accounted for by means of an attraction operator at the top of the tree (see section 3.2.6). The postcyclic rules of Question Formation and Fronting perform the actual attraction process by this operator.

In the implementation, they are grouped into their own postcyclic subgrammar, together with, again, the rules of Tree-Pruning:

%Grammar: Attraction
--> Rule: Question-Formation (Once,,)
--> Rule: Fronting (Once,,)
--> Rule: AUX-Attraction (Once,,)
--> Grammar: Tree-Pruning
--> Stop

We shall first have a look at Question Formation. The attraction process takes place in two steps. In the first step, the question operator at the top of the (sub)tree attracts a wh-constituent if one is present in its scope, or it adopts a yes/no-question marker. Formally, this step is defined as:

(Seuren 1996, p.96)

For wh-questions, this could be implemented as follows:

--> SD: (S ... # (X "QUE") # ...1 # (<WH> ...))2 #
--> COND: ^{x1 x2: ... (NP (S ... (<WH> ...))})
--> SC: #1 #2 #3 #4
     ==> #1 #2 << &x2 #3

For yes/no-questions, it could be implemented as: 61

--> SD: (S ... # (X "QUE") # ...1 )
--> COND: ^{x1 : ... (<WH> ...))} & (x2 = "(Y/N-0)")
--> SC: #1 #2
     ==> #1 #2 << &x2

In fact, we combine these two implementations into one rule:

%Rule: Question-Formation
--> SD: (S ... # (X "QUE") # ...1 # (<WH> ...))2 # ...3 )
--> COND: {{x1 x3: ... (<WH> ...))} ->
     {x1 x3: ... (NP (S ... (<WH> ...))}) &
     (x2 is_empty -> (x2 = "(Y/N-0)"))
--> SC: #1 #2 #3 #4 ==> #1 #2 << &x2 #3

---

61 For the time being, we only consider main questions and leave embedded questions out of account. There is also no implementation yet of Seuren's observation that Question Formation can cause the deletion of the complementizer of embedded clauses: "If the attracted c[WH] is the subject or predicate nominal of a complement non-NP S. cons[that] is obligatorily deleted" (Seuren 1996, p.105-6).
This rule looks for a question operator ("X"QUE"). If there is a \(wh\)-constituent ((\(<WH>\)...)), which is not embedded in an \(NP\)[S] (x1x3: \(...\)(NP(S...(\(<WH>\)...))), then this constituent is detached and right-adopted by the operator (#2 \(\ll\) x2). If there is no \(wh\)-constituent (x2 is empty), then an empty \(yes/no\)-marker is adopted by the operator instead (x2 = "(Y/N=0)").

Next, let us have a look at the largely analogous first step of Fronting. Seuren's formal definition is:

\[ \text{"x}[\text{Foc}] \text{ right-adopts any major constituent } c[\text{Foc}], \text{ except S, /S, AUX, and } \textit{not}. \text{ X is } c[\text{Foc}]/c[\text{Foc}].\]  

(ibid. p.96)

This first step of Fronting is basically implemented as follows:

%Rule: Fronting
\[\rightarrow SD: (S \ldots \# (X "Foc") \ldots \# (\langle Foc \rangle \ldots ) \#)\]
\[\rightarrow \text{COND: None}\]
\[\rightarrow \text{SC: #1 #2 #3 #4 } \Rightarrow \text{ #1 #2 } \ll \text{ #4 #3}\]

Seuren adds to his definition: "If \(c[\text{Foc}]\) is a subject \textit{that}-clause dummy \(NP[\text{it}]\) is deleted" (1996, p.96). We incorporate this into the implemented rule:

\[\rightarrow SD: (S \ldots \# (X "Foc") \ldots 1 \# (NP "it")?2 \ldots 3 \# (\langle Foc \rangle \ldots ) 4 \#)\]
\[\rightarrow \text{COND: } \{\{x1 \ x3; \ldots \ (NP "it")\}\} \&\]
\[\{\{x4: (\langle af<\text{SU}> \ldots \ (COMP "that") \rightarrow (x2 = "") \} \&\]
\[\text{no identical trees}\]
\[\rightarrow \text{SC: #1 #2 #3 #4 #5 #6 } \Rightarrow \text{ #1 #2 } \ll \text{ #6 #3 \&x2 #5}\]

Now, the rule checks whether there is a dummy NP present ((NP "it")?2) and removes this node ((x2 = "")) if the fronted constituent is a subject \textit{that}-clause ((x4: (\langle af<\text{SU}> \ldots \ (COMP "that")\)).

With main questions and with Fronting, there is a second step if the attracted constituent is not the subject or, in the case of questions, if a \(yes/no\)-marker was adopted in the absence of a \(wh\)-constituent. In both cases, this second step involves the attraction of the AUX-area.

This is implemented as one rule, which we have called AUX-Attraction. The basic form of the implementation is:

%Rule: AUX-Attraction
\[\rightarrow SD: (S \ldots \# ((X ==) (\langle af<\text{SU}> \ldots )) \ldots \#)
\[\langle V (\langle AFF<\text{AUX}> ==) \ldots (\langle AUX ==) \# \Rightarrow \# \ldots \rangle \#\]
\[\rightarrow \text{COND: None}\]
\[\rightarrow \text{SC: #1 #2 #3 #4 #5 #6 } \Rightarrow \text{ #1 #2 } \ll \text{ ( #4 #6 ) #3 #5}\]

This rule looks for an attraction operator that has adopted a node which is \textit{not} the subject (( (X ==) (\langle af<\text{SU}> \ldots ))). It singles out the AUX-area
\((\text{V} \ (\text{AFF}<\text{AUX}> \ ==) \ ... \ (<\text{AUX}> \ ==))\), which is then right-adopted by the attraction constituent \#2 \ \ll \ \{\#4 \ \#6\}.\(^{62}\)

To make sure that the rule selects the whole AUX-area, which occasionally consists of three lexical nodes rather than two (e.g. when not is incorporated by AUX), we add a condition:

\[
\rightarrow \text{SD:} \ \{S \ ... \ # \ (X ==) \ (<^af\text{SU}> \ ...)) \ # \ ...
(S<\text{SLASH}> \ ... \ #
(V \ (\text{AFF}<\text{AUX}> \ ==) \ ... \ (<\text{AUX}> \ ==) \ # \ ==>1 \ # \ ...2) \ #
\rightarrow \text{COND:} \ ^{\{x1 \ x2: \ ... \ (<\text{AUX}> \ ...)}\}
\]

Also, the spine-features need to be adjusted:

\[
\rightarrow \text{COND:} \ ^{\{x1 \ x2: \ ... \ (<\text{AUX}> \ ...)}\} \ \&
(^{\langle x1 \text{ is empty} \rangle} \ \text{->} \ x1 \text{ plus } "^{<\text{spine}>"})
\rightarrow \text{SC:} \ #1 \ #2 \ #3 \ #4 \ #5 \ #6 \ ==> \ #1 \ #2 \ \ll \ \{\ #4 \ #6 \ }<\text{spine} \ \#3 \ \& x1
\]

Let us address the issue of the categorial status of the attraction operator and of the nodes that arise due to the adoption. In section 3.2.6, I have explained that the SA category of both attraction operators is V\text{Attr}. But what about their surface category? Seuren introduces a special type of surface category for these operators “of the form X/Y, meaning “takes an X to become a Y”” (1996, p.104-5). In other words, the surface category determines what kind of constituent is to be attracted. What makes this category type special is that it depends on the structural configuration of the rest of the tree. That is, in main clauses the surface category of the question operator is c[WH]/c[WH] (meaning “takes a wh-constituent to become a wh-constituent”) if the attracted wh-constituent is the subject of the clause; if it is not, the surface category is c[WH]/AUX/S (meaning “takes a wh-constituent to become AUX//S, which then takes AUX to become /S”).

We prefer not to follow the formalization in this respect. First of all, it is an unnecessary extension of the notational apparatus. As I showed above, it is perfectly well possible to trigger Question Formation a and b without calling upon the surface category of the operator. The main reason for Seuren to introduce this notation is to express the similarity of the processes involved in Question Formation and Fronting. In our view, the implementation offers enough other ways to reflect this similarity. That is, in the Formation Rules the operators are generated in exactly the same way, and in the Postcycle the attraction rules are grouped into their own subgrammar (with the second step of Question Formation and of Fronting even being performed by one and the same rule, AUX-Attraction).

Another ground for not following this aspect of the formalization is that it deviates, in our view, too much from the concept of surface category in the rest of the grammar. In all other cases, the surface category is something which is given in the

---

\(^{62}\) Note that the string referred to by \#6 only consists of the closing brackets needed to make the AUX-area a proper constituent.
lexicon and only retrieved from it in a fairly late phase of the generation process. It is, in a sense, truly superficial. In contrast, in Seuren’s treatment of the attraction operators, the surface category suddenly becomes something which drives the generation process by triggering a rule. It becomes an element that doesn’t quite seem to fit in with the rest of the machinery.

However, having established that we do not need the complex surface categories that Seuren presupposes, we still need to assign a category to the operator and its mother node(s). For the operator itself, it does not seem to be very essential what category it has. It is an abstract element, so let us assign it an abstract category, say X. But what about the complex nodes that arise through the adoption? What is their status? Normally, with adoption, the arising node is of the same category as the adopting node, so in this case the category of the complex nodes would be X too. However, it seems more reasonable to say that they are of the same status as the adopted constituent, rather than the adoptive node. For example, if a wh-NP is attracted, the resulting constituent should also be an NP. That is, instead of the operator, we should perhaps consider the attracted node to be the head node.

Tentatively, we could account for that in the implemented rule by applying left adoption by the attracted node instead of right adoption by the operator (but leaving the resulting constituent in the same place as the operator): 63

%Rule: Question-Formation
--> SC: #1 #2 #3 #4 --> #1 #2 >> &x2 #3

%Rule: Fronting
--> SC: #1 #2 #3 #4 #5 #6 --> #1 #2 >> #6 #3 &x2 #5

%Rule: AUX-Attraction
--> SC: #1 #2 #3 #4 #5 #6
    --> #1 { #2 >> (#4 #6)<spine> }<spine> #3 &x1

Let me illustrate the effect of these rules with an interrogative version of our example sentence, as given in (58).

(58)  Who did John seem to have kissed?

63 This is possible because in GRAMSY the operation of adoption is separated from the place where the resulting constituent ends up in the tree: we do not have to leave the adopting node in its original place to obtain a valid SC.
Question-Formation ==>  

AUX-Attraction, Tree-Pruning-1 ==>
4.4.5 Do-Support

The next postcyclic rule in the English grammar is called Do-Support and is not only responsible for the well-known phenomenon of the insertion of dummy verb *do* in certain configurations, but also for the deletion of the simultaneous tense operator in other configurations. Seuren’s definition is as follows:

“For any V-cluster: O not followed directly by a V-node filled by a surface lexical verb is replaced by the dummy verb *do*. Otherwise v(O) is deleted, in which case redefine AUX according to AUX-DELIMITATION.”

(Seuren 1996, p.96)

Thus, the rule Do-Support replaces the simultaneous tense operator with dummy *do* in the negative sentence (59a), but removes this operator in our example sentence, repeated here as (59b). 64

(59)  a  John did not seem to have kissed Mary.
       b  John seemed to have kissed Mary.

In the implementation, the rule again has its own subgrammar, since its application may also trigger Tree-Pruning:

%Grammar: Do-Support
  --> Rule: Do-Support
  --> Grammar: Tree-Pruning
  --> Stop

Implementing the two effects separately, we would arrive at the following two rules:

%Rule: Do-Support-1
  --> SD: (S ... (V ... #
         (V "SIM") # ...1)
  --> COND: ^{x1: { (V )? (V ===) }}
  --> SC: #1 #2 ==> #1 (V "do")

%Rule: Do-Support-2
  --> SD: (S ... (V ... #
         (V "SIM") # { (V )? (V ===) ...})
  --> COND: None
  --> SC: #1 #2 ==> #1

The first rule looks for a simultaneous tense operator ("SIM") and if it is not followed by a lexical verb (^{x1: { (V )? (V ===) }}), it is replaced by the dummy verb ("do"). If the tense operator is followed by a lexical verb, on the other hand, the second rule simply removes it (#1 #2 ==> #1).

These rules can be simply integrated into one rule, with two alternative SCs:

---

64 Recall that the earlier rule of SIM-Deletion also removes the simultaneous tense operator, but only if the operator is followed by the verb be (see section 4.4.2). Furthermore, note that also the abstract operator EMHP may stand in between the tense operator and the lexical verb, and thus result in supportive *do*, as in John DID seem to have kissed Mary.
%Rule: Do-Support

\[ \rightarrow\text{SD: } (S \ldots (V \ldots \# (V \"SIM\") \# \{(V )? (V ==)?1 \ldots2)} \]

\[ \rightarrow\text{COND: } x_1 \text{ is}\_\text{empty} \rightarrow \wedge(x_2: \{(V )? (V ==)} \]

\[ \rightarrow\text{SC: } \#1 \#2 \]

\[ \Rightarrow [x_1 \text{ is}\_\text{empty}] \#1 (V \"do\") \]

\[ \Rightarrow [\wedge(x_1 \text{ is}\_\text{empty})] \#1 \]

Furthermore, the AUX-feature of the removed operator needs to be copied onto the dummy verb do or onto the lower lexical verb:

\[ \rightarrow\text{SD: } (S \ldots (V \ldots \# (V\{3 \"SIM\") \# \{(V )? (V ==)?1 \ldots2)} \]

\[ \rightarrow\text{SC: } \#1 \#2 \#3 \#4 \]

\[ \Rightarrow [x_1 \text{ is}\_\text{empty}] \#1 (V\{4\&E3} \"do\") \#3 \#4 \]

\[ \Rightarrow [\wedge(x_1 \text{ is}\_\text{empty})] \#1 \#3 \#4\&E3> \]

The effect of this rule on our example structure is as follows:

```
      S
     /\  /
    NP  S\SLASH>
      \ /\
       V /\V
        AFF V\AUX> V V NP
          \ /\ \
          V Prt V
          \ /\ \
          V V
          \ /\ \
          AFF V

John PAST SIM seem to have EN kiss Mary
```

Do-support, Tree-Pruning-1

```
      S
     /\  /
    NP  S\SLASH>
      \ /\
       V /\V
        AFF V\AUX> V V NP
          \ /\ \
          V Prt V
          \ /\ \
          V V
          \ /\ \
          AFF V

John PAST seem to have EN kiss Mary
```
4.4.6 Affix Handling

The rule of Affix Handling is intermediary between syntax and morphology, in the sense that it transforms right-branching syntactic structures into left-branching morphological units. Seuren’s formal definition is the following:

“For any node A—cat=Aff and A—rb=L (L is a lexical constituent), or A—rb=B and B—d=L-
C or C=L (B and C are not lexical constituents or C is a lexical constituent but on a spine): L
RIGHT-ADOPTS A”

(Seuren 1996, p.61)

In the implementation, Affix Handling, too, has its own subgrammar. From this subgrammar, also the rules of Tree-Pruning and Surface Category Assignment (see p.103-104) are called. The latter will change the labels of the morphological units according to Seuren’s specification: “V over Aff[PRES/PAST] is relabelled ‘FV’ (finite verb). V over Aff[EN] is relabelled ‘PaP’. V over Aff[ING] is relabelled ‘PrP’.” (ibid. p.96).

%Grammar: Affix-Handling
---> Rule: Affix-Handling
---> Grammar: Tree-Pruning
---> Rule: Surface-Category-Assignment

The basic implementation of the rule is very simple:

%Rule: Affix-Handling
---> SD: {S ... (V # (AFF ==) # {(V )}? # (V ==) #
---> COND: None
---> SC: #1 #2 #3 #4
     ===> #1 #3 #4 << #2

Additionally, the spine and AUX features need to be adjusted:

---> SC: #1 #2 #3 #4
     => #1 #3 { #4<spine> << #2<spine> )<^AUX,^spine>

In order to trigger the rule of Surface Category Assignment with the right surface labels, we enhance the implemented rule as follows:

---> SD: {S ... (V # (AFF ==) # {(V )}? # (V ==) #
---> COND: (((x1: "PRES") | (x1: "PAST")) --> (f2 = "<FV>")) &
           ((x1: "EN") --> (f2 = "<PaP>")) &
           ((x1: "ING") --> (f2 = "<PRP>"))
---> SC: #1 #2 #3 #4
     => #1 #3 { #4<spine> << #2<spine> )<^SCA<&f2>,^AUX,^spine>

Also, a minor adaptation to the rule of Surface Category Assignment is needed to exclude the morphological units from the special treatment of multi-layered V-clusters (see also p.103-104):
Surface-Category-Assignment:

\[ \Rightarrow SD: (S \ldots # (V<SCA>1 \ldots)2 # \]

\[ \Rightarrow COND: \{(x2: (V \ == 3)) \& \{(?&LEX: x3): (V<surf><4>) \}\ |
\{(x2: (V<SCA><4> ( \ldots )5 ( \ldots )6 )) \&
\{(^{f4} \ features "<FV|PAP|PRP>" \rightarrow x5 \ plus "<SCA>" \) \&
\{(x3 ='(x5 x6)')) |
\{(x2: (V \ ... 3)) \& (f4 = "<V>") \}) \&
\{(x3 = (x3 node\_category\_change f4)) \&
\{(x3 plus "<s1,^SCA,^type,^rul>" \)
\end{align*}

\[ \Rightarrow SC: \#1 \#2 \Rightarrow \#1 \& x3 \]

The effect of two applications of Affix Handling, each followed by Tree Pruning and Surface Category Assignment, is:

Affix-Handling, Tree-Pruning-1, Surface-Category-Assignment \[ \Rightarrow \]

Affix-Handling, Tree-Pruning-1, Surface-Category-Assignment \[ \Rightarrow \]
4.4.7 Mince

The final rule of the Postcycle is Mince. It is a more or less cosmetic rule, which flattens out the tree structure by cutting up the V-cluster and re-attaching its subconstituents directly under the S-node. Formally:

"Detach the highest subconstituent on the spine of any C-cluster. Re-attach the detached subconstituent to the right of its parent. Never detach structure belonging to AUX or to a C-island or to a lexical constituent." (Seuren 1996, p.96)

The rule is, again, placed into its own subgrammar together with the associated rules of Tree Pruning.

%Grammar: Mince
--> Rule: Mince
--> Grammar: Tree-Pruning
(Once,,)

The basic implementation is as follows:

%Rule: Mince
--> SD: (S ... (&C ... # (&C<spine> ...)) # ...
--> COND: None
--> SC: #1 #2 #3 --> #1 #3 #2<spine>

This rule looks for a C-cluster and detaches its highest subconstituent on the spine ((&C<spine> ...)) to place it to the right of its mother. To incorporate the ban on mincing AUX-structures, we add the following condition:\(^{65}\)

--> SD: (S ... (&C ... #1 # (C<spine> ...))2 # ...
--> COND: {x2: {x2: {x2? {AUX}> ^{x1: ... (<AUX>)}}}

The effect of the rule on our example structure is the following:

\(^{65}\) The protection of C-islands and lexical constituents from being minced has not been implemented yet.
The resulting tree is the Surface Structure of our example sentence.

## 4.5 Conclusion

To conclude this chapter, I will summarize what we claim to have contributed to the model of Semantic Syntax by implementing the transformational grammar for English. As we have looked exclusively at English in this chapter, only the first of our two research questions as formulated in chapter 1 is relevant here. Recall that, with respect to the descriptive adequacy of Semantic Syntax, we were interested to know whether the grammars as formalized by Seuren (1996) are internally consistent and account for exactly the set of sentences that they are claimed to account for. Additionally, from a technolinguistic viewpoint, we wanted to establish if Seuren’s formalization is a descriptively optimal representation of the underlying theory. The other question, regarding Semantic Syntax’ crosslinguistic adequacy, will be addressed in the next chapter.

I think it is fair to say that by and large we have succeeded in implementing the Semantic Syntax transformational grammar for English satisfactorily. We have managed to build a principle-based grammar that remains very close to Seuren’s formalization and that generates the surface structures of practically all the sentences that Semantic Syntax is claimed to cover so far. I will now refine this general con-
clusion and give a balanced synopsis of the various challenges we have met with respect to the model’s descriptive adequacy.

As I said, our implementation generates practically all of Seuren’s sentences. This means there are a few constructions the generator does not account for:

- First of all, we have not implemented the lowering of quantifiers (see section 4.3.2). Although we do not expect there to be any principled problem with implementing the process in itself, the precise effect and conditions are so much intertwined with the internal grammar of noun phrases that it seems to make no sense implementing quantifier lowering without having also a good noun phrase grammar.

- Another, related process that we have not implemented is the so-called Scope Ordering Constraint (see footnote 45, p.98). Although Seuren (1996, p.300-309) theorizes about this constraint on the surface ordering of scope-sensitive elements rather extensively, he does not integrate it into the formalized rules. Thus we have not implemented it either.

- A third phenomenon our implementation does not account for completely is the embedded question (see footnote 53, p.118 and footnote 61, p.126). Although we have accomplished a satisfactory implementation of the formation rules with respect to embedded questions (see section 3.2.6), the relevant transformational rules do not handle them correctly at present. Although it seems feasible to make the necessary adjustments to these rules (Complementizer Insertion, Question Formation and AUX-Attraction), we have chosen not to do so for the time being in order not to complicate them too much.

- For the same reason, we have not implemented Seuren’s observation that the complementizer of an embedded clause is deleted if a subject or predicate nominal of this clause is attracted by a higher attraction operator (see also footnote 61, p.126).

- Furthermore, what our implementation does not account for either is the placement of adverbs in sentences where the AUX area is cluster-final (see footnote 60, p.125). Seuren’s definition of Adverb Placement holds a condition to ensure that in such sentences the adverb does not end up after the verb. This condition has not been integrated into our implementation of the rule, although again we see no principled problem with it.

- Finally, what we have also left unimplemented is the condition on Mince that excludes C-islands and lexical constituents from being split up.

Aside from these points, the implementation generates all of the sentences that Seuren claims the Semantic Syntax model covers. Now, let us look at the differences between Seuren’s formalization and our implementation in how these sentences are accounted for.
Looking at the general structure of the grammar, the first difference that strikes the eye is the Precycle. Although Seuren assumes there to be a precyclical component, it is not part of his formalization. In contrast, the implementation does make use of such a component. The implemented grammar Precycle holds a number of rules that take care of various mechanisms and processes specified by Seuren implicitly or explicitly in the theory. Thus, in our generator, the Precycle holds a filter rule and nine tree-editing rules. The filter rule in fact combines the PPI-filter and the EMPH-filter that Seuren describes as two separate restrictions on SA-structure into one rule (see section 4.2.1). The nine tree-editing rules fall in two separate categories with somewhat different relations to Seuren’s model. The rules Argument-Functions, New-Sentence-Unit, Rest-of-Sentence-Unit, Spine, and Lookup-Rule-Features are all implementations of notions that are explicitly defined by Seuren, albeit not as separate rules (see sections 4.2.4 to 4.2.7). The other four rules are specific to the implementation and have no direct counterparts in Seuren’s theory: Affix-to-feature-bundle and Spine are cosmetic rules that have to do with the transition from GRAMGEN to GRAMTSY, NP-pruning is a provisional rule that will become obsolete once a sound internal grammar of NP’s is realized, and Perfective-Auxiliary is a rule needed because our implementation of the formation rules deviates from the formalization66 (see sections 4.2.2, 4.2.3 and 4.2.8).

In contrast, the rules in the other two implemented grammars (Cycle and Postcycle) all have their counterparts in the formalization. Still, there are some general differences between the implemented rule system and the formalized one (see the introduction to section 4.3). Some of the rules in our Cycle are not independent rules in the formalization but rather subroutines that are part of larger rules, such as Tree Pruning and Surface Category Assignment. We have chosen to treat these routines as rules in their own right, because they can be seen as independent transformations. In our view, they differ fundamentally from another one of Seuren’s subroutines called Adoption, which cannot be separated from the larger routines that it is involved in. Adoption is not a corollary or side effect to a certain transformation, it is a type of transformation. Therefore, unlike the other subroutines, it is no independent rule in the implementation. Also, we do not distinguish between local versus global routines, as Seuren does. Our implementation tool GRAMTSY only allows us to define local processes. It has made us redefine Seuren’s global routines as local rules (which are only called when other routines may have created a situation in which they are applicable) and thus reduce the number of elementary operation types. As a result, the rule system of Semantic Syntax has become simpler and more unified. Another distinction that has become obsolete in our implementation is that between structurally induced routines and category-induced routines: in the GRAMTSY formalism, all rules are essentially structurally induced. The only distinction that remains meaningful is between the lexically induced rules versus the

---

66 The grounds for having this rule will become clearer in the grammars for Dutch and German (see section 5.5.2).
structurally induced ones – these each have received their own subgrammars in the implementation. A downside to the GRAMTSY tool is that it forces us to impose an explicit rule ordering, which is for the most part arbitrary (but see footnote 41, p.92). Furthermore, we have chosen to implement Seuren’s corollaries as cyclic rules (with the exception of AUX-delimitation), although Seuren seems to regard them as non-cyclic routines. The advantage of doing so is that we can now treat them on a par with Seuren’s subroutines of Tree Pruning and Surface Category Assignment, which we have also implemented as separate cyclic rules. Finally, a rather superficial difference is the nomenclature of some of the rules: instead of language-specific names like To-Insertion, It, That-Insertion and O-be-Deletion, we use names like Particle-Insertion, Extraposition, Complementizer-Insertion and SIM-Deletion.

Besides these general differences, some of the rules in the Cycle and Postcycle also differ from their formalization counterparts on more specific aspects. Let me now, to conclude this chapter, recapitulate these specific differences. First, we concentrate on the cyclic rules:

- Within every rule involving adoption, we remove all features from the newly created mother-nodes, although Seuren never explicitly states this to be necessary (see footnote 42, p.94). However, if we would not do this, the consecutive adoption transformations would result in a huge proliferation of features.

- The rules of Past Participle and Present Participle have been implemented as one rule, because of their similarity (see section 4.3.1). Also, part of Seuren’s definition of these rules turned out to be best treated as a special condition to the postcyclic rule of Affix Handling instead.

- For Tree Pruning, the implementation differs in three respects from the formalization (see section 4.3.3): in the implementation, each of the Tree-Pruning varieties constitutes a separate rule; Tree-Pruning 3 needs an extra condition to ensure that \( v_p[S] \)-nodes are not pruned; and Tree-Pruning 4 is not included in the implementation because the relevant configuration never arises in the present set of implemented rules.

- With Surface Category Assignment, we had to make a specific provision for branching nodes: whereas Seuren describes only in general terms that the routine is to assign phrase status to a branching node, we have the rule of Object Incorporation add a special feature to the arising branching node that states its correct surface category; next, the rule of Surface Category Assignment picks up this feature as the surface category to be assigned (see sections 4.3.4 and 4.3.8).

- With Subject Raising, we have not implemented the restriction that it sometimes applies only in subject or object position, which Seuren indicates by means of subscripts to the rule feature (see footnote 46, p.105).

- With Particle-Insertion, one of Seuren’s conditions is that the /S-constituent that is to receive the particle to must not contain AUX; since the AUX-features
are not available in the Cycle, however, we have translated this into the condition that the /S-constituent must not contain a finite tense affix. Also, we have found that Seuren’s definition of the rule does not take into account the possibility that the to-adopting V-node is the only V under /S and we have adapted the rule to correct this (see section 4.3.6).

- With Subject Deletion, we have only implemented the vertical variant. Horizontal Subject Deletion is limited to prepositional predicates followed by a gerund and since such constructions are included in the set of sentences that Seuren claims to account for, we have not included it in the implementation.

- The implementation has made us realize that the rules of Subject Deletion and Object Incorporation could both be treated as structurally induced instead of lexically induced, which would mean a higher level of generalisation since the relevant lexical rule features would no longer be needed. We have not carried this observation through in the implementation however, because it would lead to an ordering paradox with our separate subgrammars for lexically and structurally induced rules (see footnotes 49, p.110 and 50, p.112).

- With Extraposition, we have made a provisional implementation of the relevance of the topic-comment modulation of a sentence to the optionality of this rule, even though Seuren only theorizes about this relevance and has not incorporated it into the formalization (see section 4.3.9).

- A major difference to the formalization is our implementation of Copula Insertion. In section 4.3.10, I have argued that our technolinguistic approach has brought to light a conceptual problem in Seuren’s formalization of Copula Insertion: his version of the rule constitutes a violation of the principle of strict cyclicity since it affects the tree structure outside the present cyclic domain. We have found a relatively simple solution to this, by replacing Seuren’s two-step process with a one-step process. An important additional advantage of this approach is that it renders the rule of Predicate Raising superfluous for the English grammar. Also, we have not implemented two extra conditions on Copula Insertion as mentioned by Seuren, since they seem to be always met automatically in the languages and constructions under consideration.

- The rule Complementizer Insertion as implemented operates at a different stage than its counterpart in the formalization. The condition that the finite S must be embedded is in fact something that can only be checked in the next higher cycle, so we have redefined the rule to apply at this higher level (see section 4.3.11). As indicated in footnote 54, p.118 this might alternatively be taken as evidence that the rule should be regarded as a postcyclic rule instead, but this is something we have not investigated further.

Specific differences between implementation and formalization in the Postcycle are the following:
• Because of the changed view on level 2 adverbials that we have developed in chapter 3 (see section 3.2.5), we have taken new definitions of AUX Delimitation and Adverb Placement as the basis for our implementation (see footnotes 55, p.120 and 59, p.123).

• In the implementation, the AUX-area is marked by means of a feature on the lexical nodes that belong to the area. We have chosen not to make use of Seuren’s defining notions of entrance and exit, because the latter of these is somewhat problematic with cluster-final AUX-areas (see footnote 56, p.121).

• We have decided not to re-call the rule of AUX-Delimitation in case of a possible redefinition of the AUX-area. Instead, this redefinition is handled by the rules that induce it (SIM-Deletion and Do-Support, see sections 4.4.2 and 4.4.5). What should be investigated, though, is whether reversal of the order of the rules AUX-Delimitation and SIM-Deletion is possible, as this would relieve the need for such a redefinition of AUX by SIM-Deletion (see footnote 58, p.123).

• With Question Formation and Fronting, we have reached an important improvement of Seuren’s formalization (see section 4.4.4). The similarity between these two rules is expressed by Seuren through a special category type of the form X/Y. We, on the other hand, have separated the construction-specific first step of the operation (Question Formation and Fronting) from the second step (AUX-Attraction), which is identical for the two constructions. Partly because of this, we could do away with the special category type and thus avoid this unwanted extension of the notational apparatus. We have tried to establish what the surface category of the relevant nodes should be instead, but have not found fully satisfying solution. We have arrived, however, at a tentative implementation with an alternative kind of adoption (see footnote 63, p.129).

• With Affix Handling, the rule of Surface Category Assignment is re-called to relabel the morphological units (see section 4.4.6). In Seuren’s formalization, this is part of the rule of Affix Handling itself.

In the next chapter, we will take a look at the transformational grammars of Dutch and German to gain more insight in the crosslinguistic adequacy of the Semantic Syntax model.
CHAPTER 5  The transformational grammar for Dutch and German

5.1  Introduction

In the preceding chapter, I discussed the implementation of the transformational grammar for English. For the other two languages under consideration, Dutch and German, we need to adapt some of the subgrammars and rules. For this purpose, we make use of the parametrization mechanisms offered by GRAMTSY (see also Figure 4.1 on p.76). I will discuss all adaptations in the following sections.

5.2  Grammar parametrization

GRAMTSY offers us the possibility to parametrize both the grammars and the rules. Parametrizing the grammars means that we specify a general grammar with parameters, plus a specific parameter setting file for each language. GRAMTSY then compiles a language-specific grammar file on the basis of these. The parameter settings determine which rules and subgrammars are to be included in the language-specific grammar. In our case, this is useful for rules that do not apply in all three languages, but only in one or two of them.

Only two of the rules that were described above apply exclusively in English, namely the postcyclic rules of Adverb-Placement (see section 4.4.3) and Do-Support (see section 4.4.5). Thus, the subgrammar Postcycle is parametrized as follows:

%Grammar: Postcycle
   --> Rule: AUX-Delimitation
   --> Grammar: SIM-Deletion
   [SOMETIMES] --> Grammar: Adverb-Placement
   --> Grammar: Attraction
   [SOMETIMES] --> Grammar: Do-Support
   --> Grammar: Affix-Handling
   --> Grammar: Mince
   --> Stop

The parameter [SOMETIMES] means that the rule or subgrammar is by default not included in the language-specific grammar, unless the parameter file states that it is to be included. The following statements in the English parameter file do precisely this:
Besides rules that only apply in English, there are also a few rules that only apply in Dutch and/or in German. The cyclic rule Predicate Raising and the postcyclic rules Particle-Participle and V-Final apply both in Dutch and in German. The postcyclic rule End Cluster Arrangement is exclusive to the Dutch grammar; there are no rules exclusive to the German grammar. I will discuss each of these rules in the following subsections. A survey of the grammar parametrization is given in Table 5.1.67

Table 5.1 Grammar parametrization in GENIUS.

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Rule</th>
<th>Parameter</th>
<th>English</th>
<th>Dutch</th>
<th>German</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle</td>
<td>Predicate-Raising</td>
<td>OFTEN</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Postcycle</td>
<td>Adverb-Placement</td>
<td>SOMETIMES</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Do-Support</td>
<td>SOMETIMES</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Particle-Participle</td>
<td>OFTEN</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>V-Final</td>
<td>OFTEN</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>End-Cluster-Arrangement</td>
<td>SOMETIMES</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

5.2.1 Predicate Raising

Seuren (1996, p.247-249) argues that, whereas the English complementation system exclusively depends on Subject Raising and Subject Deletion, Dutch and German S-complements undergo Predicate Raising or Subject Deletion.68 That is, the Dutch and German equivalents (60) and (61) of our example sentence involve raising of the lower predicate instead of the lower subject.69

(60) Jan scheen Marië te hebben gekust.70
     John seemed Mary to have kissed.

---

67 In sections 5.4.2 and 5.5.5, it will become clear that Dutch and German each need an extra rule that is language-specific: precyclic Directionality-Switch in Dutch and the cyclic corollary Particle-Insertion-2 in German. These rules have no independent counterparts in the formalization. Also, Dutch and German need an extra calling of the rules Argument-Functions compared to English (see section 5.2.1).
68 In Seuren’s formalization, Predicate Raising is also triggered in the English auxiliary system by the copular verb be. In the implementation, however, the rule of Copula Insertion has been altered for independent reasons (see section 4.3.10), with the side effect of rendering the rule of Predicate Raising superfluous. Thus, in English, Predicate Raising is only reflected in lexicalized combinations like let go, but the emergence of these combinations does not belong to overt syntax (see also Seuren 1996, p.55).
69 Note that the rule of Subject Raising is still needed in the grammars for Dutch and German, because it is part of the tense routine in the auxiliary system.
70 A synonymous and equally grammatical variant is Jan scheen Marie gekust te hebben, with the past participle in front of the te-infinitive (as in German). This variability is explained in section 5.2.4.
(61) Jan schien Marie geküsst zu haben.
John seemed Mary kissed to have.

We extend the cyclic subgrammar Lexically-Induced with a parametrized rule as follows:

%Grammar: Lexically-Induced
   --> Rule: Subject-Deletion (Once,,)
   --> Rule: Subject-Raising (Once,,)
   [OFTEN] --> Rule: Predicate-Raising (Once,,)
   --> Rule: Object-Incorporation (Once,,)
   --> Rule: Participle (Once,,)
   --> Rule: Lowering-to-S (Once,,)
   --> Rule: Lowering-to-V (Once,,)
   --> Rule: Lowering-to-the-Right (Once,,)
   --> Stop

The parameter [OFTEN] means that the relevant rule or subgrammar is by default included in the language-specific grammar, unless the parameter file states that it is not to be included. Since the rule Predicate Raising should be part of both the Dutch and the German Cycle, but not of the English Cycle, the parameter file for English is provided with the following statement:

%Grammar: Lexically-Induced
[ ] Rule: Predicate-Raising (Once,,)

Seuren formally defines Predicate Raising as follows:

"For any node $S_c = d = V \alpha-S_c^a \beta$ or $S_e = d = V \alpha-S_e^a \beta$ where $V, rulf = <PR>$ ($\alpha, \beta$ are possibly null strings of nodes): (a) detach the node $N \rightarrow cat = V, N \rightarrow p = S_d$ or $S_a$, (b) the node $V \rightarrow p = S_e$ $\alpha$-adopts $N$, (c) SCA and AF feature re-assignment as required." (Seuren 1996, p.64)

It receives the following basic implementation:71

%Rule: Predicate-Raising
   --> SD: (S # (V$rulf<PR>) ...)) # <==$
          (S$<$type<FIN>> # (V ...)) #
   --> COND: None
   --> SC: #1 #2 #3 #4
      ==> #1 #2<$rulf<PR>> <= #4$spine> #3

This rule looks for a predicate with the relevant triggering feature ((V$rulf<PR>) ...)) and has it right-adopt the predicate of its argument-S (#2 <= #4). This argument-S may have either zero or one tense operator, but not two ((S$<$type<FIN>>)). The triggering feature is removed from the adopting predicate (#2$<$rulf<PR>>)) and a spine feature is added to the raised predicate (#4$spine>).

An automatic side effect of removing the lower predicate out of its own S is what Seuren calls Downgrading 2:

---

71 Actually, this implementation is specifically for the right-branching clusters of Dutch. For German we need another SC (a form of rule parametrization), which will be discussed in section 5.4.1.
“For any node X such that X.cat=S or /S and X.d=α (α may be Φ; no node of α dominates V
with no S or at most one /S in between): (a) erasure of X, (b) re-attachment of α higher up.”
(Seuren 1996, p.58)

This is integrated into the implemented rule in the following way:

```
--> SD: (S # (V<rulf<PR> ...) # <== #
          (S<^type<FIN>> # (V ...)) # ==> # ) #
---> SC: #1 #2 #3 #4 #5 #6 #7
==> #1 #2<^rulf<PR>> << #5<spine> #3 #6
```

Now, the argument-S that loses its V-node (#4) is erased from the tree, together
with its closing bracket (#7). The remaining daughters of the S-node (#6) are attac-
hed higher up.

With some predicates, the application of Predicate Raising is optional. This is inte-
grated into the implemented rule by means of an alternative SC, which outputs an
extra tree for the relevant predicates:

```
--> SD: (S # (V<rulf<PR>1> ...) # <== #
          (S<^type<FIN>> # (V ...)) # ==> # ) #
---> SC: #1 #2 #3 #4 #5 #6 #7
==> #1 #2<^rulf<PR>> << #5<spine> #3 #6
       [f1 features "<OPT>"] #1 #2<^rulf<PR>> #3 #4 #5 #6 #7
```

If the inducing predicate is marked for optional Predicate Raising ((V<rulf<PR>1>...) with f1 features "<OPT>"), then a second output
tree is generated in which no raising takes place at all (#1 #2 #3 #4 #5 #6 #7). The only change is the removal of the inducing rule feature
(#2<^rulf<PR>>).

As stated in Seuren’s formal definition of Predicate Raising, the application of this
rule also involves Surface Category Assignment and Argument Feature re-
assignment. Thus, we add a triggering feature for Surface Category Assignment to
the raised node:

```
--> SC: #1 #2 #3 #4 #5 #6 #7
==> #1 #2<^rulf<PR>> << #5<SCA, spine> #3 #6
```

To arrive at the re-assignment of argument functions, we re-call the rule Argument-
Functions (see 4.2.4), which was also called in the Precycle:

```
%Grammar: Corollaries
[FREQUENCY] --> Rule: Argument-Functions (Once,)
---> Grammar: Tree-Pruning (Once,)
---> Rule: Particle-Insertion (Once,)
---> Rule: Surface-Category-Assignment (Once,)
---> Rule: That-Insertion (Once,)
---> Rule: Cyclic-Cosmetics (Once,)
---> Stop
```

Since the re-assignment of argument functions as a corollary of Predicate Raising
only occurs in Dutch and German (of the three languages under consideration), we ex-
clude it from the English subgrammar in the parameter file for the latter language:
%Grammar: Corollaries
[ ] Rule: Argument-Functions

We trigger the rule of Argument Functions by having the rule of Predicate Raising remove the \(<af>\) feature:

\[
\begin{align*}
\text{SD: } & (S \# (V <rulf<PR>> 1 \ldots) \# \text{\texttt{==2}} \# (S \# \text{type<FIN>> \# (V \ldots) \# \text{\texttt{==>}} \# ) \# ) \# ) \\
\text{COND: } & (\ldots (x2 \text{ is empty} \rightarrow x2 \text{ plus } "<af>") \ldots) \\
\text{SC: } & \text{\texttt{=} } 1 \# 2 \# 3 \# 4 \# 5 \# 6 \# 7 \\
\text{implies } & \text{\texttt{=} } 1 \# 2 <rulf<PR>> \# 5 <SCA, spine> \# x2 \# 6
\end{align*}
\]

The rule of Argument Functions itself should be adjusted as follows:

%Rule: Argument-Functions
\[
\begin{align*}
\text{SD: } & \ldots (S (V \ldots) 1) \# \\
& (\&ARG<af, ^SLASH> \ldots) \# (\ldots (NP \ldots)?2 (\&ARG \ldots) 3) \# \ldots) \\
\text{COND: } & (\ldots (x3 \text{ is empty} \rightarrow x3 \text{ plus } "<af<DO>>") \& \\
& (\ldots (x2 \text{ is empty} \rightarrow (x2 \text{ plus } "<af<IO>>") \& \\
& (\ldots (x3 \text{ is empty} \rightarrow ((x2 \text{ plus } "<af<DO>>" \& x3 \text{ plus } "<af>>")))) \\
\text{SC: } & 1 \# 2 \# 3 \Rightarrow 1 \# 2 <af<SU>> \# x2 \& x3
\end{align*}
\]

In the Cycle, the rule is triggered by the absence of an argument function on the first argument (<\text{\texttt{af}}\text{>}), the feature that is removed by the rule of Predicate Raising. On the other two arguments, the rule does not simply add a feature bundle, but rather replaces it by means of the exclamation mark (<\text{\texttt{af}}\text{>}). Furthermore, the rule does not apply when the first argument is a downgraded clause (<\ldots^SLASH\ldots>).

The effect of the rule and its corollaries on the relevant cycle of sentence (60) is as follows.\textsuperscript{72}

\[
\text{\texttt{S}} \quad \begin{array}{c}
V<rulf<PR>> \\
S<type<PERF>> \\
\end{array} \\
\begin{array}{c}
V \\
NP<af<SU>> \text{NP}<af<DO>> \\
\end{array} \\
\begin{array}{c}
V \\
AFF \ldots V \\
\end{array} \\
schijnen \begin{array}{c}
\text{hebben EN} \\
kussen \\
Jan \text{ Marie}
\end{array}
\]

\textsuperscript{72} Note that there is no re-assignment of the argument functions because the complement clause is the only argument of the predicate raising verb, so the disappearance of the S-node through downgrading does not change the number of arguments. The subject and direct object of the raised predicate automatically become the subject and direct object of the resulting verb cluster.
5.2.2 Particle-Participle

The three languages each involve application of Affix Handling (see section 4.4.6). The morphological units that arise through Affix Handling are subsequently assigned their correct surface categories by application of Surface Category Assignment (see section 4.3.4). In Dutch and German, the existence of verbal particles (for example, op in Dutch opeten as in sentence (62) and auf in German aufessen) makes this process slightly more complex: if a verb-affix unit that is to be assigned the category of PaP (past participle) has a brother node labelled Prt (verbal particle), the parent V must also be relabelled PaP (see Seuren 1996, p.225-226 and p.283).\(^7\)

(62) De kat heeft de muis opgegeten.
The cat has the mouse up-eaten

(=The cat has eaten the mouse.)

We integrate this into the implementation by means of an additional rule of Particle-Participle within the subgrammar Affix Handling. The general grammar file is extended with a rule that applies after Affix Handling itself but before Surface Category Assignment:

\[
\begin{align*}
\text{%Grammar: Affix-Handling} & \rightarrow \text{Rule: Affix-Handling} \\
& \rightarrow \text{Grammar: Tree-Pruning} \\
\text{[OFTEN]} & \rightarrow \text{Rule: Particle-Participle} \\
& \rightarrow \text{Rule: Surface-Category-Assigment}
\end{align*}
\]

The English parameter file is provided with a statement to exclude the rule call:

\[
\text{%Grammar: Affix-Handling} \\
[-] \text{Rule: Particle-Participle}
\]

\(^7\) According to Seuren (1996), this particle is to the right of the verb in Dutch, but to the left of it in German. However, Seuren (1997, p.28-29) has independent reasons for eliminating this difference to the effect that in both languages the particle is always to the left of the verb (which corresponds to the surface order); see also section 5.2.4.
The rule is implemented as follows:

%Rule: Particle-Participle
--> SD: (S ... # ( # (Prt ===) (<SCA<PAP>> ...))
--> COND: None
--> SC: #1 #2 ==> #1 #2<!SCA<PAP>>

This rule looks for a node that is to become a past participle ((<SCA<PAP>> . . . )), with a particle as its left brother ((Prt ===)). The SCA-triggering feature together with the category specification is copied from the participle node onto its mother (#2<!SCA<PAP>>).

At the relevant stage in the generation of sentence (62), the effect of this rule is the following:

```
S
   | NP | S<SLASH>
   |    |
   | V  | S   |
   |    |
   | V  | NP |
   |    |    |
   | V  | V  |
   |    |    |
   | V  | AFF   |
   |    |    |
   | V  | PRT  |
   |    |    |
   | V   |
   |    |
   | AFF |
   |    |
   | x kat x hebben PRES op eten EN :x muis x
```

Particle-Participle, Surface-Category-Assignment (2x) ==> 

```
S
   | NP | S<SLASH>
   |    |
   | V  | S   |
   |    |
   | V  | NP |
   |    |    |
   | V  | V  |
   |    |    |
   | V  | AFF   |
   |    |    |
   | V  | PRT  |
   |    |    |
   | V   |
   |    |
   | PAP |
   |    |
   | V  |
   |    |
   | AFF |
   |    |
   | :x kat x hebben PRES op eten EN :x muis x
```

5.2.3 V-Final

Another postcyclic rule that applies in Dutch as well as in German, but not in English, is V-Final. This rule is responsible for moving the verbal cluster partly (in main clauses) or completely (in subordinate clauses) out of its post-subject (or post-fronted-element) position to the far right, with some restrictions. Seuren's definition of the rule is as follows:

"V-Final (applies to the non-AUX part of the V-cluster in main clauses, and to the whole V-cluster in non-main S or /S): Move the (sub)cluster to the far right of /S, but never across an embedded S or /S. It may stop before a Prep3Phrase (not Prep3)." (Seuren 1996, p.226)
The difference between English on the one hand and Dutch and German on the other is illustrated in (63) and (64). In (63a) and (64a), the English verbal cluster is in its original position before the direct object. In (63b) and (63c), the non-finite part of the Dutch or German verbal cluster is moved to the far right; in the subordinate clauses (64b) and (64c), the whole Dutch or German verbal cluster is moved to the right.

(63)  
   a) John has kissed Mary.  
   b) Jan heeft Marie gekust.  
   c) Jan hat Marie geküsst.

(64)  
   a) It seems that John has kissed Mary.  
   b) Het schijnt dat Jan Marie heeft gekust / gekust heeft.\textsuperscript{74}  
   c) Es scheint, dass Jan Marie geküsst hat.

In the implementation, we add the rule to the general grammar file, together with a parameter and its own subgrammar:

%Grammar: Postcycle
   --> Rule: AUX-Delimitation
   --> Grammar: SIM-Deletion
[SOMETIMES] --> Grammar: Adverb-Placement
   --> Grammar: Attraction
[SOMETIMES] --> Grammar: Do-Support
   --> Grammar: Affix-Handling
[OFTEN] --> Grammar: V-Final
   --> Grammar: Mince
   --> Stop

%Grammar: V-Final
   --> Rule: V-Final  \hspace{1cm} \textit{(Once, ,)}
   --> Grammar: Tree-Pruning

In the English parameter file, we state that the subgrammar V-Final does not apply in this language:

%Grammar: Postcycle
[++] Grammar: Adverb-Placement
[++] Grammar: Do-Support
[--] Grammar: V-Final

The basic implementation of the rule for main clauses is as follows:

%Rule: V-Final:
   --> SD: (S ... (S<SLASH> # (V # (FV ...)) # <= ) # ==> # )1
   --> COND: ^{x1: (S<unit<^0>>)}
   --> SC: #1 #2 #3 #4 #5 ==> #1 #3 #5 #2 #4

This rule looks for a V-cluster ((V (FV...) <=)) within a main clause (^{x1: (S<unit<^0>>)})\textsuperscript{75}. The non-finite part of this cluster is moved to the far right.

\textsuperscript{74} The two alternative verb orderings are explained in section 5.2.4.
To include also the correct processing of subordinate clauses, the rule must be extended as follows:

\[ \rightarrow SD: (S \ldots (S<\text{SLASH}> \# (\&V \# (FV \ldots)) \# 1 \# \langle= \rangle \# \rangle \# 2) \]
\[ \rightarrow COND: ((\{x2: (S<\text{unit}^<\text{0}>)) \& x1 \text{ is empty}) \]
\[ (^{\langle=\rangle}x2: (S<\text{unit}^<\text{0}>)) \& (x1 \text{ is empty}) \]

Now, the condition part of the rule takes care of the different treatment of the two types of clauses. In main clauses \(^{\langle=\rangle}x2: (S<\text{unit}^<\text{0}>))\), the finite verb is to remain at its original position so we separate it from its containing cluster \((FV\ldots)?1 \text{ with } (x1 \text{ is empty})\). In subordinate clauses \({x2: (S<\text{unit}^<\text{0}>))\), the finite verb, which may or may not be present, is to be moved along with the rest of the cluster so we do not separate the two \((x1 \text{ is empty})\). Since in subordinate clauses, the moved cluster may also consist of just the finite verb, a variable \(\&V\) is used for its label; this variable stands for either \(V\) or \(FV\).

To have the verbal cluster stop obligatorily before an embedded clause and optionally before a level 2 prepositional phrase, we extend the rule again:

\[ \rightarrow SD: (S \ldots (S<\text{SLASH}> \# (\&V \# (FV \ldots)) ?1 \# \langle= \rangle \# \rangle \# 3) \#
\[ \rightarrow COND: ((\{x6: (S<\text{unit}^<\text{0}>)) \& x1 \text{ is empty}) \]
\[ (^{\langle=\rangle}x6: (S<\text{unit}^<\text{0}>)) \& (x1 \text{ is empty})) \]
\[ (\{x2: \ldots (S<\text{7}) \rightarrow 
\[ ^{(\langle=\rangle}x4 \text{ is empty}) \rightarrow (x5 \text{ is empty})) \]
\[ (\{x3 \text{ is empty}) \]
\[ \rightarrow SC: \#1 \#2 \#3 \#4 \#5 \#6
\[ \rightarrow \#1 \#3 \#5 \#2 \#4 \#6
\[ \rightarrow \][^\langle=\rangle\#3 \#5 \#2 \#4 \#6]

Now, if the clause contains an embedded clause \((\{\{NP\}?4(S\ldots)\})\?5\) with \(^{\langle=\rangle}x4 \text{ is empty}) \rightarrow (x5 \text{ is empty}))\), the moving cluster does not cross it. If the clause contains a level 2 prepositional phrase \((PP\langle=\rangle<\text{ADV2}>)\ldots)?3 with \(^{\langle=\rangle}x3 \text{ is empty}))\), two tree structures emerge: one with the V-cluster before the PP, and one with the cluster after the PP.

Finally, we add a condition to exclude vacuous movements:

\[ \rightarrow COND: (\{\{x6: (S<\text{unit}^<\text{0}>)) \& x1 \text{ is empty}) \]
\[ (^{\langle=\rangle}x6: (S<\text{unit}^<\text{0}>)) \& (x1 \text{ is empty})) \]
\[ (\{x2: \ldots (S<\text{7}) \rightarrow 
\[ ^{(\langle=\rangle}x4 \text{ is empty}) \& (x2 \text{ is empty}) \& (x3 \text{ is empty})) \]
\[ (^{\langle=\rangle}x4 \text{ is empty}) \rightarrow (x5 \text{ is empty})) \]
\[ (\{x3 \text{ is empty}) \]
\[ \rightarrow SC: \#1 \#2 \#3 \#4 \#5 \#6
\[ \rightarrow \#1 \#3 \#5 \#2 \#4 \#6
\[ \rightarrow \][^\langle=\rangle\#3 \#5 \#2 \#4 \#6]

\(^{34}\) Recall that main clauses either have a zero unit feature or no unit feature at all, depending on whether or not they contain a subordinate clause (see section 4.2.5).
The rule has the following effect on the tree structure of the Dutch sentence (64b). Note that it only applies in the subordinate clause; the verbal cluster of the main clause in this sentence is not moved because, first of all, it consists of nothing more than the finite verb and, secondly, it would not be moved across the embedded clause anyhow.

5.2.4 End-Cluster-Arrangement

In Dutch, the verbal cluster is not only subject to V-Final but it often also allows several alternative internal orderings. This was already shown above for a cluster containing a past participle in example (64b), repeated here:

(64) b Het schijnt dat Jan Marie heeft gekust / gekust heeft.
The particle of compound verbs, too, can occupy various places within the cluster, as is illustrated in (65). The variations can be combined, so clusters with both a past participle and a particle have even more variants, as shown in (66). Notice that the past participle may not precede the particle, however, which is why (66g) is ungrammatical.

(65) a  ... dat de kat de muis zou willen opeten.
     ... that the cat the mouse would want up-eat
b  ... dat de kat de muis zou op willen eten.
c  ... dat de kat de muis op zou willen eten.

(= ... that the cat would want to eat the mouse.)

(66) a  ... dat de kat de muis moet hebben opgegeten.
     ... that the cat the mouse must have up-eaten
b  ... dat de kat de muis moet opgegeten hebben.
c  ... dat de kat de muis opgegeten moet hebben.
d  ... dat de kat de muis moet op hebben gegeten.
e  ... dat de kat de muis op moet hebben gegeten.
f  ... dat de kat de muis op moet gegeten hebben.
g  * ... dat de kat de muis gegeten moet op hebben.

(= ... that the cat must have eaten the mouse.)

A postcyclic rule called End-Cluster-Arrangement is responsible for these permutations. It seems closely related to V-Final, so we include it in the subgrammar V-Final. It only applies in Dutch, so it receives the [SOMETIMES]-parameter. A second parameter ensures that it applies optionally:

```
%Grammar: V-Final
--- Rule: V-Final (Once,)
[SOMETIMES]  --- Rule: End-Cluster-Arrangement (,,optional)
--- Grammar: Tree-Pruning
--- Stop
```

The following statement is added to the Dutch parameter file:

```
%Grammar: V-Final
[+] Rule: End-Cluster-Arrangement
```

Seuren (1996) gives a fairly complex definition of the rule, involving the assignment of weights (ranging from 0 to 3) to the various parts of the verbal cluster, and the obligatory or optional raising of lower-weighted elements across the higher-weighted ones. Seuren (1997), however, accounts for the same facts by means of a revised, much simpler version of the rule. This revision is possible thanks to two new assumptions regarding, first, the existence of left-branching V-clusters in Dutch (bringing their analysis more in line with the analysis of the German clusters; see also section 5.4) and, second, the underlying form of compound forms being particle-verb (op-eten) instead of the more artificial order verb-particle (eten-op, see also footnote 73, p.148). The revised definition runs as follows:
"Verbal particles and past participles may climb through a V-cluster without limit, but a past participle may never climb across a verbal particle belonging to the same verb."

(Seuren 1997, p.29)

The climbing of particles and participles receives the following implementation:

\[
\text{%Rule: End-Cluster-Arrangement}
\]

\[
\text{---&gt; SD: } \{S \ldots (V \ldots \# (\ldots)) \# \{(V)\}? # (\ldots)1 \#}
\]

\[
\text{---&gt; SC: } \{x1: (VPrt ==\ldots)\} | \{x1: (PAP \ldots)\}
\]

\[
\text{---&gt; SC: } \#1 \#2 \#3 \#4
\]

\[
\text{---&gt; SC: } #1 \text{ (V #4<^spine> #2<spine>) #3}
\]

This rule looks for a verbal particle or a past participle (\{(x1: (VPrt ==\ldots)\} | \{(x1: (PAP \ldots)\)}) within a V-cluster. If this particle or participle has a node to its left within the same cluster, the order of the two is reversed (\{(V #4 #2\)). The spine features are updated, since the branching directionality of the cluster remains unchanged.

Seuren’s condition that past participles may never climb across their own particles seems to be unnecessarily strict. As far as we know, Dutch has no compound verbs that trigger Predicate Raising and may have a perfect tense in their scope, so there can never emerge a cluster in which a past participle might climb across a particle belonging to another verb.\(^76\) Hence, the condition can be simplified to: "a past participle may never climb across a verbal particle". This is easily added to the implementation:

\[
\text{---&gt; SC: } \{x1: (VPrt ==\ldots)\}
\]

\[
\text{---&gt; SC: } \{x2: (VPrt ==\ldots)\} | \{x2: (PAP \ldots)\} & ^{x1: (VPrt ==\ldots)}\}
\]

Now, if the moved element is a participle (\{(x2: (PAP \ldots)\}), the element it moves across cannot be a particle (\^{x1: (VPrt ==\ldots)}\}).

The rule has the following effect on the tree structure of (64b). Since the rule applies optionally, both of the trees below (the one affected by End-Cluster-Arrangement, as well as the unaffected one) will be input to the next subgrammar, Mince.

\(^76\) The only possible candidate that comes to mind is the verb nalait("refrain from"). This is a compound verb (\ldots dat ik dit na heb gelaten, meaning "that I have refrained from this") which might be analysed as inducing Predicate Raising since a sentence like \ldots dat ik dit naliei te doen ("that I refrained from doing this") appears to be acceptable to some speakers of Dutch, although the Subject Deletion variant \ldots dat ik naliei dit te doen is far more natural. It is impossible, however, to have a perfect tense in the scope of naliei, i.e. \ldots dat ik dit naliei te hebben gedaan / \ldots dat ik naliet dit te hebben gedaan ("that I refrained from having done this") is ungrammatical.
5.3 Lexical rule parametrization

Besides parametrizing the grammars, GRAMTSY also allows us to parametrize the rules. This can be done in two ways. First, we can parametrize a rule in essentially the same way as a grammar, that is, by prefixing part of the rule (a condition, for example) with a parameter like \[\text{SOMETIMES}\] or \[\text{OFTEN}\]. Just as with the grammar parametrization, plus and minus symbols in the parameter file indicate whether the relevant part of the rule is applicable in the language at hand.

A second way of parametrization is achieved by means of variables that have different values in different languages. These variables are between square brackets, too, and are used, for example, to generalize over specific lexical fillers that vary across languages. I will refer to these variables as parameters as well, to avoid any confusion with the grammar-internal variables.
Besides distinguishing these two different techniques of rule parametrization, we can also make a conceptual classification. Thus, in the present and following sections, I will first discuss the lexical parametrization, then the branching parametrization, and finally any remaining adjustments to the rules.

### 5.3.1 Lexicon lookup

All rules that involve lexicon lookup or a direct reference to lexical fillers need to be parametrized. For the lexicon lookup, we replace the grammar-internal variable &LEX by the parameter [ @LEX ]. This is done in all rules that refer to the lexicon: Argument-Functions, Lookup-Rule-Features, Perfective-Auxiliary et cetera. Each language’s parameter file specifies the value for this global parameter, i.e. the name and location of the lexicon file. So, for example, the Dutch parameter file contains the following statement:

%Global:
LEX: ../lexicon/dutch2

### 5.3.2 Lexical fillers

Parametrization of rules that refer directly to lexical fillers is taken care of in essentially the same way. Let me illustrate this by means of the rule SA-filter (see section 4.2.1). Recall the implementation we have arrived at for English:

%Rule: SA-Filter
--> SD: {S ... (V<ADV2> ===1) (S<ADV2> ...2)
--> COND: ([(x1: "not") & (x2: (V<ADV2> {"not" | "EMPH"}))]) |
{x1: "EMPH")
--> FILTER: Tree filtered out because of level 2 adverb below &x1

The literal string “not” needs to be replaced by a parameter to have the rule operate correctly in Dutch and German as well.\(^7\) So, we adapt the condition of the rule:

--> COND: ([(x1: [@NEG]) & (x2: (V<ADV2> {[@NEG] | "EMPH"})])) |
{x1: "EMPH")

Furthermore, we add a parameter value to each language’s parameter file:

**English:**
%Rule: SA-Filter
NEG: "not"

\(^7\) Note that the lexical filler “EMPH” is an abstract element, which will be the same in all languages, so it does not need to be replaced by a parameter. We do assume it is present in the grammars of Dutch and German as well, although Seuren (1996) has not included it in his formalization for these two languages (presumably because in these languages it has no counterpart effect of the English phenomenon of Do-support, see also section 4.4.5).
A number of other rules contain similar literal strings that we replace by parameters in exactly the same way. Table 5.2 lists all these rules, with the parameters and their respective values in each of the three languages.

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Rule</th>
<th>Parameter</th>
<th>English</th>
<th>Dutch</th>
<th>German</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precycle</td>
<td>SA-Filter</td>
<td>NEG</td>
<td>&quot;not&quot;</td>
<td>&quot;niet&quot;</td>
<td>&quot;nicht&quot;</td>
</tr>
<tr>
<td></td>
<td>Perfective-</td>
<td>HAVE</td>
<td>&quot;have&quot;</td>
<td>&quot;hebben&quot;</td>
<td>&quot;haben&quot;</td>
</tr>
<tr>
<td></td>
<td>Auxiliary</td>
<td>BE</td>
<td>--</td>
<td>&quot;zijn&quot;</td>
<td>&quot;sein&quot;</td>
</tr>
<tr>
<td>Cycle</td>
<td>Extraposition</td>
<td>DUMMY</td>
<td>&quot;it&quot;</td>
<td>&quot;het&quot;</td>
<td>&quot;es&quot;</td>
</tr>
<tr>
<td></td>
<td>Copula-Insertion</td>
<td>COP</td>
<td>&quot;be&quot;</td>
<td>&quot;zijn&quot;</td>
<td>&quot;sein&quot;</td>
</tr>
<tr>
<td></td>
<td>Particle-Insertion</td>
<td>TO</td>
<td>&quot;to&quot;</td>
<td>&quot;te&quot;</td>
<td>&quot;zu&quot;</td>
</tr>
<tr>
<td></td>
<td>That-Insertion</td>
<td>THAT</td>
<td>&quot;that&quot;</td>
<td>&quot;dat&quot;</td>
<td>&quot;dass&quot;</td>
</tr>
<tr>
<td>Postcycle</td>
<td>Fronting</td>
<td>DUMMY</td>
<td>&quot;it&quot;</td>
<td>&quot;het&quot;</td>
<td>&quot;es&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>THAT</td>
<td>&quot;that&quot;</td>
<td>&quot;dat&quot;</td>
<td>&quot;dass&quot;</td>
</tr>
</tbody>
</table>

What is striking in this listing is, obviously, that some of the parameters occur in more than one rule, with the same language-specific values. For example, the parameter [@THAT] is apparently needed in both the cyclic rule That-Insertion and the postcyclic rule Fronting. In this light, it seems more economical to regard all these parameters as global instead of rule-specific parameters.

Additionally, there appears to be enough reason to regard the parameters for the copular verb [@COP] and the perfective auxiliary [@BE] as instances of the same parameter too. After all, it is probably not a coincidence that both functions are fulfilled by be/zijn/sein in all three languages (although the use of be as an auxiliary does not occur anymore in modern English, see section 5.5.2); for one thing, the same parallel can be found in the Romance languages too. Thus, subsuming these two instances under the more general denominator [@BE], we can reduce the total number of lexical parameters to six, as shown in Table 5.3.
Table 5.3 Revised lexical rule parametrization in GENIUS, by means of global parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>English</th>
<th>Dutch</th>
<th>German</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEG</td>
<td>&quot;not&quot;</td>
<td>&quot;niet&quot;</td>
<td>&quot;nicht&quot;</td>
</tr>
<tr>
<td>HAVE</td>
<td>&quot;have&quot;</td>
<td>&quot;hebben&quot;</td>
<td>&quot;haben&quot;</td>
</tr>
<tr>
<td>BE</td>
<td>&quot;be&quot;</td>
<td>&quot;zijn&quot;</td>
<td>&quot;sein&quot;</td>
</tr>
<tr>
<td>DUMMY</td>
<td>&quot;it&quot;</td>
<td>&quot;het&quot;</td>
<td>&quot;es&quot;</td>
</tr>
<tr>
<td>TO</td>
<td>&quot;to&quot;</td>
<td>&quot;te&quot;</td>
<td>&quot;zu&quot;</td>
</tr>
<tr>
<td>THAT</td>
<td>&quot;that&quot;</td>
<td>&quot;dat&quot;</td>
<td>&quot;dass&quot;</td>
</tr>
</tbody>
</table>

5.4 Branching parametrization

A central notion within Semantic Syntax is branching directionality. Seuren (1996, 1997) accounts for many word order differences between languages by assuming different branching directionalities in (parts of) their grammars. Each (sub)tree is either left- or right-branching and this directionality is represented by means of the notion spine (see also section 4.2.6).

In the four European languages that Seuren (1996) describes, the SA-structures are right-branching throughout. That is, the arguments are always to the right of their predicate. That this is not the case in all languages, however, is shown by Seuren in a tentative Semantic Syntax account of Turkish, with left-branching SA-structures. In all these languages, Turkish as well as English, Dutch, German and French, the morphological structures are predominantly left-branching – that is to say, they all inflect their verbs by means of suffixes rather than prefixes.

In English and French, the syntax remains right-branching throughout: Lowering always involves left-adoption, and Predicate Raising always involves right-adoption. In German, on the other hand, the V-clusters that arise through successive raising and lowering operations are, in principle, left-branching. Only certain verbs, called R-verbs, make the cluster right-branching under certain conditions (see Seuren 1996, p.271-280; Seuren 1997; and Richter 2000). Dutch has an intermediary position between English and French on the one hand, and German on the other: its V-clusters are, in principle, right-branching, but a few so-called L-verbs make the cluster optionally switch to left-branching (see Seuren 1997). The branching directionalities for the three languages under consideration are summarized in Table 5.4.
Table 5.4 Branching directionality in the three languages.

<table>
<thead>
<tr>
<th>Part of the grammar</th>
<th>English</th>
<th>Dutch</th>
<th>German</th>
</tr>
</thead>
<tbody>
<tr>
<td>SA-structure</td>
<td>right</td>
<td>right</td>
<td>right</td>
</tr>
<tr>
<td>V-clusters</td>
<td>right</td>
<td>right, except for L-verbs</td>
<td>left, except for R-verbs</td>
</tr>
<tr>
<td>Morphology</td>
<td>left</td>
<td>left</td>
<td>left</td>
</tr>
</tbody>
</table>

Why does Seuren assume these different branching directionalities in the V-clusters of English, Dutch and German? In English and most Dutch V-clusters, the order of the verbs is the same as the hierarchical order of these verbs in the SA-structure. That is, the semantically highest verb also surfaces as the first verb in the cluster – see, for example, (67a,b). This indicates that the branching directionality of the V-cluster is the same as in the SA-structure, namely right-branching, as illustrated in Figure 5.1.

![Diagram](image)

Figure 5.1 English/Dutch: from a right-branching SA to a right-branching V-cluster.

(67)  
 a  It is true that I have (1) gone (2) dancing (3).
 b  Het is waar dat ik ben (1) gaan (2) dansen (3).
 c  Es ist wahr, dass ich tanzen (1) gegangen (2) bin (1).

In German, on the other hand, the order of the verbs is reversed in most cases. That is, the semantically highest verb surfaces as the last verb in the cluster – see (67c). This can be accounted for by going from a right-branching SA-structure to a left-branching V-cluster, as illustrated in Figure 5.2.

---

78 The numerical notation indicating the semantic scope of the verbs relative to each other was introduced by Bech (1955) and adopted by Seuren (1997) and Richter (2000). It is also used in traditional, structuralistic grammars like Haeseryn et al. (1997).
How is the implementation to be adapted to cater for these left-branching clusters? This I will discuss in the next subsection. The rationale behind the Dutch L-verbs and German R-verbs, as well as their implementation, will be the topic of a separate subsection (section 5.4.2).

### 5.4.1 Left-branching German clusters

First, we need to change the cyclic rules that build up the verbal cluster; in the German grammar, these rules must involve right-adoption instead of left-adoption, and left-adoption instead of right-adoption. And second, the SD’s of various postcyclic rules must be adapted to recognize left-branching clusters instead of right-branching ones.

The cluster-building rules are Lowering to V (see section 4.3.2), Predicate Raising (see section 5.2.1), Participle (see section 4.3.1) and Copula Insertion (see section 4.3.10). To arrive at left-branching clusters in German, we parametrize the SC’s of these rules. In a right-branching syntax, the adoption is to the left of the V-constituent in the rules of Participle, Lowering to V and Copula Insertion, so in a left-branching syntax it should be to the right.80

---

79 In this connection, Seuren (1996, 1997) only mentions Lowering and Predicate Raising, but certainly also Participle is a rule that builds up the V-cluster by means of adoption. Copula Insertion is a cluster-building rule in our implementation, because we have incorporated the adoption into it to make it a one-step process.

80 Recall that we have integrated the rules Past Participle and Present Participle into one implemented rule called Participle. Although strictly speaking the rule of Present Participle is not needed in the rule systems for Dutch and German (as they do not contain a progressive auxiliary to trigger it), this is no reason for adjusting the implemented rule. The triggering rule feature simply does not occur, so the rule will never be applied wrongly.
%Rule: Participle
--> SC: #1 #2 #3
[EITHER RB] ==> #1 #2<^rulf<PAP,PRP>> (#x2 >> #3<spine>)<^type,^spine>
[OR LB] ==> #1 #2<^rulf<PAP,PRP>> (#3<spine> << #x2)<^type,^spine>

%Rule: Lowering-to-V
--> SC: #1 #2 #3 #4
[EITHER RB] ==> #1 #3 (#2<SCA,^rulf<L>> >> #4<spine>)<^spine,^type>
[OR LB] ==> #1 #3 (#4<spine> << #2<SCA,^rulf<L>>)<^spine,^type>

%Rule: Copula-Insertion
--> SC: #1 #2
[EITHER RB] ==> #1 (#x2 >> #2<spine,SCA>)<^type,^spine,^SCA>
[OR LB] ==> #1 (#2<spine,SCA> << #x2)<^type,^spine,^SCA>

With the rule of Predicate Raising, the adoption is exactly the other way around:

%Rule: Predicate-Raising
--> SC: #1 #2 #3 #4 #5 #6 #7
[EITHER RB] ==> #1 #2<^rulf<PR>> << #5<SCA,spine> &x2 #6
[OR LB] ==> #1 #5<SCA,spine> >> #2<^rulf<PR>> &x2 #6

With the [EITHER/OR]-parameter, the [EITHER]-case is always the default. So, for each rule, we add a statement to the German parameter file that it should be left-branching instead:

%Rule: Participle
[+] [LB]

%Rule: Lowering-to-V
[+] [LB]

%Rule: Copula-Insertion
[+] [LB]

%Rule: Predicate-Raising
[+] [LB]

The effect of the left-branching versions of Participle and Lowering-to-V, for example, can be illustrated with the relevant cycle in the generation of sentence (67c) on p.159:
A postcyclic rule that is sensitive to the branching-direction of the verbal cluster is AUX-Delimitation (see p. 120-122). Recall our implementation for English (and Dutch):

```
%Rule: AUX-Delimitation
--> SD: (S ... (V # (AFF<^AUX> {"PRES"|"PAST"}) ...) # (V ==) #
--> COND: None
--> SC: #1 #2 #3 ==> #1 #2<AUX> #3<AUX>
```

In the left-branching clusters of German, the affix has the adjacent lexical V-node to its left instead of its right. Hence we parametrize the SD of this rule in much the same way as we parametrized the SCs of the cluster-building rules above:

```
--> SD: (S ...
[EITHER RB] (V # (AFF<^AUX> {"PRES"|"PAST"}) ...) # (V ==) #
[OR LB] # (V ==) {}) # (AFF<^AUX> {"PRES"|"PAST"}) #
```

Again, the [EITHER]-case is the default, so we add a statement to the German parameter file that it should be left-branching instead:

```
%Rule: AUX-Delimitation
[+][LB]
```

The effect of the left-branching version of AUX-Delimitation can again be illustrated with sentence (67c):
The postcyclic rule AUX-Attraction (see section 4.4.4) is sensitive to the order of the constituents in the verbal cluster, too. Recall our implementation for English (and Dutch):

% Rule: AUX-Attraction

--> SD: (S ... # ((X ==) (<<af<SU>> ...)) # ... (S<SLASH> ... #
     (V (AFF<\auX> ===) ... (<<\auX>> ==)) # ==1 # ...2) #
--> COND: (^ (x1 is empty) -> x1 plus "<<\auX>>") &
     ^[x1 x2: ... (<<\auX>> ...)]
--> SC: #1 #2 #3 #4 #5 #6
     ==> #1 { #2 >> { #4 #6 }<\auX> }<\auX> #3 &x1

Parametrizing this rule for branching directionality is more complicated and affects both the SD and the COND-part:
Now, we must state in the German parameter file that it should be left-branching and not right-branching:

%Rule: AUX-Attraction
[+] [LB]
[-] [RB]

Obviously, also the rule of Affix Handling (see section 4.4.6) is sensitive to the V-cluster's branching directionality, since in a left-branching cluster the affixes are to the right of the lexical verbs that they are to be attached to. The rule has been implemented for English (and Dutch) as follows:

%Rule: Affix-Handling

---> SD: (S ... (V # (AFF ==l) # {(V )? # (V ==) #
---> COND: {((x1: "PRES") | (x1: "PAST")) -> (f2 = "<FV>")} &
{((x1: "EN") -> (f2 = "<FAP>")) &
{((x1: "ING") -> (f2 = "<PRP>"))
---> SC: #1 #2 #3 #4
---> #1 #2 #3 { #4<spine> << #2<spine> }<SCA<&f2>, ^AUX, ^spine>

One could argue that, in the left-branching clusters of German, Affix Handling is not needed since the affixes are already on the right-hand side of the lexical verbs. However, there are two reasons for assuming this rule to be part of the German grammar as well. First, although the present or past tense affix is on the right-hand side of its verb, the two of them do not form a constituent; rather, the affix is part of the cluster just as any other lowered predicate. So the rule of Affix Handling is needed to change the constituency of the elements, even though their order remains unaltered. And second, Affix Handling has the side effect of assigning the correct category label to the lexical constituent by triggering the corollary Surface Category Assignment. Parametrizing the rule of Affix Handling involves both its SD and SC:

---> SD: (S ... [EITHER RB] (V # (AFF ==l) # {(V )? # (V ==) #
---> COND: {((x1: "PRES") | (x1: "PAST")) # {((V ==) # {(#4<spine> ==1) #
---> SC: #1 #2 #3 #4
[OR LB] #1 #3 #4<spine> << #2<spine> #2<spine> #2<spine> #3

We add a statement to the German parameter file that it should be left-branching:

%Rule: Affix-Handling
[+] [LB]

Let me illustrate the effect of the left-branching version of AUX-Delimitation with the same German sentence again:
A fourth postcyclic rule that is sensitive to the branching directionality of the V-cluster is V-Final (see section 5.2.3). Recall our implementation for Dutch:
%Rule: V-Final

\[
\rightarrow \text{SD: (S ... (S<SLASH> # (S V # (FV ...))?1 # <= ) #}
\]

\[
\rightarrow \text{COND: (1) } \{ \text{([x6: (S<unit<0>>) & x1 is_empty) |}
\]

\[
\rightarrow \text{SC: #1 #2 #3 #4 #5 #6}
\]

\[
\rightarrow \text{[} (x1 is_empty) | (x2 is_empty) \text{]} #1 #3 #5 #6 #2 #4
\]

\[
\rightarrow \text{[^}(x4 is_empty) -> (x5 is_empty))}
\]

Only the SD-part must be adjusted to the right-branching clusters of German:

\[
\rightarrow \text{SD: [S ... [S<SLASH> #}
\]

\[
\rightarrow \text{[EITHER RB] (S V # (FV ...))?1 # <= ) #}
\]

\[
\rightarrow \text{[OR LB] (S V # (FV ...))?1 # #}
\]

\[
\rightarrow \text{[} (x1 is_empty) | (x2 is_empty) \text{]} #1 #3 #5 #6 #2 #4
\]

Again, we add a statement to the German parameter file:

%Rule: V-Final

[+] [LB]

The rule has the following effect on the left-branching cluster of example sentence (67c):

\[
\text{V-Final, Tree-Pruning-1 } \Rightarrow
\]
The postcyclic rule of Mince (see section 4.4.7) also needs to be parametrized for branching directionality. Recall the implementation for right-branching English (and Dutch):

%Rule: Mince

\[ \text{SD: (S ... (C ... 1 # (C<spine> ...))2 # ...)) #} \]
\[ \rightarrow \text{COND: (} \{x2: (C)? \langle\text{AUX}\rangle\} \rightarrow \langle x1: ... \langle\text{AUX}\rangle\} \]
\[ \rightarrow \text{SC: #1 #2 #3} \rightarrow #1 #3 #2\langle\text{spine}\rangle \]

In the left-branching clusters of German, with the spine on the left-hand side, the detached subcluster needs to be placed to the left of its original mother-node:

\[ \text{SD: (S ... \[\text{EITHER RB]} (C ... 1 # (C<spine> ...))2 # ...)) #} \]
\[ \rightarrow \text{SC: #1 #2 #3} \]
\[ \text{[EITHER RB]} \rightarrow #1 #3 #2\langle\text{spine}\rangle \]
\[ \text{[OR LB]} \rightarrow #1 #3\langle\text{spine}\rangle #2 \]

Again, we add a statement to the German parameter file:

%Rule: Mince

[+] [LB]

The effect is as follows:
Another rule that is sensitive to the branching-directionality of the verbal cluster is SIM-deletion (see section 4.4.2), but since this rule must be changed for Dutch and German in other respects as well, I will postpone discussing its branching parametrizing to the separate subsection on this rule (section 5.5.6).

### 5.4.2 Dutch L-verbs and German R-verbs

As Seuren (1996, 1997) argues, certain verbs fall out of the regular branching pattern of their languages. In Dutch, these verbs change the directionality of the cluster to left-branching. In German, we see the exact opposite: the relevant verbs change the directionality of (part of) the cluster to right-branching. Examples are given in (68) and (69), respectively.

(68) ... dat ik dansen (2) wilde (1).
... that I dance wanted

(= ... that I wanted to dance.)
(69) ... dass ich habe (1) tanzen (3) wollen (2).
... that I have dance want

(= ... that I have wanted to dance.)

Seuren (1997) calls these verbs L-verbs and R-verbs, respectively. They are all Predicate Raising verbs, without the particle te or zu. They are all verbs with a semantic element of modality, causativity or perception (see Table 5.5). Some of them only belong optionally or marginally to the class of L/R-verbs (in the table, these are written in grey).

<table>
<thead>
<tr>
<th>type of verb</th>
<th>Dutch L-verb</th>
<th>German R-verb</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modal</td>
<td>zullen</td>
<td>--</td>
<td>shall, will</td>
</tr>
<tr>
<td></td>
<td>kunnen</td>
<td>können</td>
<td>be able, can</td>
</tr>
<tr>
<td></td>
<td>willen</td>
<td>wollen</td>
<td>want</td>
</tr>
<tr>
<td></td>
<td>moeten</td>
<td>müssen</td>
<td>must, have to</td>
</tr>
<tr>
<td></td>
<td>sollen</td>
<td>mogen</td>
<td>may</td>
</tr>
<tr>
<td></td>
<td>mogen</td>
<td>dürfen</td>
<td>be allowed</td>
</tr>
<tr>
<td>Causative</td>
<td>doen</td>
<td>lassen</td>
<td>make, cause</td>
</tr>
<tr>
<td></td>
<td>laten</td>
<td></td>
<td>let, allow</td>
</tr>
<tr>
<td>Perception</td>
<td>zien</td>
<td>sehen</td>
<td>see</td>
</tr>
<tr>
<td></td>
<td>horen</td>
<td>hören</td>
<td>hear</td>
</tr>
<tr>
<td></td>
<td>voelen</td>
<td>fühlen</td>
<td>feel</td>
</tr>
</tbody>
</table>

Although the verbs and the overall effect are roughly the same in Dutch and German, the exact details of the directionality change are different in the two languages. In Dutch, it is defined as follows:

“Dutch V-clusters are in principle right-branching, but [...] there is an optional switch to left-branching when an unclustered (except with vE[O]) lexical verb stands under a verb of the L-class standing under vE[O].”

(Seuren 1997, p.29)

Thus, the switch is optional and only takes place in clusters with two overt verbs. So, besides the left-branching (68), Dutch also has the right-branching (70) with exactly the same meaning (for more examples, see Haeseryn et al. 1997, p. 1072-1073). With more complex V-clusters, any left-branching variant is ungrammatical, as shown in (71b-d).

(70) ... dat ik wilde (1) dansen (2).
... that I wanted dance
(71) a ... dat ik heb (1) willen (2) dansen (3).  
    ... that I have want dance
b * ... dat ik heb(1) dansen (3) willen (2).
c * ... dat ik dansen (3) heb (1) willen (2).
d * ... dat ik dansen (3) willen (2) heb (1).

(= ... that I have wanted to dance.)

In German, the situation is somewhat more complicated:

"When an R-verb $V_R$ is the highest lexical verb in an S" and clustered with one or more other verbs (which are then on its left hand side) then all subsequent lowerings in the auxiliary system are right-branching, i.e. with left-attachment: (a) obligatorially when $V_R$ stands directly under a perfective auxiliary (haben or sein), in which case the rule PaP is inoperative; (b) optionally when $V_R$ stands under $v[O]$ and $v[O]$ stands under $v[werden]."$ (Seuren 1997, p.19-20)

Thus, in German the switch only takes place in clusters of three or more overt verbs: it is obligatory in clusters with haben or sein, but optional in clusters with werden. So a cluster of two verbs is always left-branching, as shown in (72a,b). Furthermore, the left-branching variant of (69) is ungrammatical, as shown in (73). Only with werden, both a left-branching and a right-branching variant exist, as shown in (74a,b).

(72) a ... dass ich tanzen (2) wollte (1).  
    ... that I dance wanted
b * ... dass ich wollte (1) tanzen (2).

(= ... that I wanted to dance.)

(73) * ... dass ich tanzen (3) wollen/gewollt (2) habe (1).  
    ... that I dance want /wanted have

(74) a ... dass ich tanzen (3) können (2) werde (1).  
    ... that I dance can will
b ... dass ich werde (1) tanzen (3) kunnen (2)  
    ... that I will dance can

(= ... that I will be able to dance.)

I will now explain how we adapt the implementation to cater for the Dutch L-verbs. In principle, the German R-verbs require similar adaptations but these are not carried through yet.

Since the appropriate configuration is easiest to recognize when no transformations have applied yet, the Precycle is a good place to identify and mark the relevant structures. Applying grammar parametrization (see p.143), we add a rule call to the relevant subgrammar in the general grammar file and include a statement in the Dutch parameter file:
The following rule identifies the relevant configurations and marks the L-verb:

%Rule: Directionality-Switch
--> SD: (S ... (V "SIM") (S # (V<rule<PR,^LB>> ==1) #
--> COND: {{?&LEX: x1}: (<LB>)
--> SC: #1 #2 ==> #1 #2<rule<LB>>

This rule looks for a Predicate Raising verb marked for left-branching in the lexicon ({{?&LEX: x1}: (<LB>)) directly below the simultaneous perfective tense operator ((V "SIM")). This verb receives a marker for left-branching (#2<rule<LB>>). Note that this rule only checks part of the L-verb condition: it does not check whether the verb to be raised is “unclustered (except with vi2[O])”. As I will explain below, this check is done within the rule of Predicate Raising itself.

The rule has the following effect on the tree structure underlying both (68) and (70):
The main rule that has to be adapted for the Dutch L-verbs is, obviously, Predicate Raising. We can use the left-branching SC we introduced for German (see section 5.4.1), with two additional conditions:

%Rule: Predicate-Raising
\[ \rightarrow \text{SD: } (S \# (V<\text{rulf<PR}>1) \ldots) \# <=2 \# \\
(S<\text{type<FIN>>} \# (V \ldots)3 \# >= \# ) \# \\
\rightarrow \text{SC: } \#1 \#2 \#3 \#4 \#5 \#6 \#7 \\
\text{[EITHER RB]} \rightarrow \#1 \#2<\text{rulf<PR>>} \#5<\text{SCA, spine}> \& \#2 \#6 \\
\text{[OR LB]} \rightarrow [f1 \text{ features} "<LB>" \& \{x3: \{(V (V "SIM") \}? (V ==\})] \\
\#1 \#5<\text{SCA, spine}> \#2<\text{rulf<PR, LB>>} \& \#2 \#6

In Dutch, for the left-branching variant to apply, the raising verb must have been marked for left-branching (f1 features "<LB>") and the raised verb ((V \ldots)3) must be unclustered except with the simultaneous tense operator (\{(V (V "SIM") \}? (V ==\)). Note that there is no need for the inverse conditions to be added to the right-branching variant, because the directionality switch is optional in Dutch.

We add a statement to the Dutch parameter file to ensure that both the right-branching and the left-branching SC apply. Also, because we do not want the conditions to apply in German, we replace them by a parameter which only receives a value in the Dutch file:

\[ \rightarrow \text{SC: } \#1 \#2 \#3 \#4 \#5 \#6 \#7 \\
\text{[EITHER RB]} \rightarrow \#1 \#2<\text{rulf<PR>>} \#5<\text{SCA, spine}> \& \#2 \#6 \\
\text{[OR LB]} \rightarrow [f1-COND] \\
\#1 \#5<\text{SCA, spine}> \#2<\text{rulf<PR, LB>>} \& \#2 \#6

%Rule: Predicate-Raising
\[ [+][\text{RB}] \\
\[ [+][\text{LB}] \\
\text{L-COND: } [f1 \text{ features} "<LB>" \& \{x3: \{(V (V "SIM") \}? (V ==\})]

Thus, in the relevant cycle, the rule outputs two structures corresponding to sentences (68) and (70):
A second cluster-building rule that must be adapted for the Dutch L-verbs is Lowering-to-V. Again, we make use of the left-branching SC designed earlier for German, with an extra parameter file:

%Rule: Lowering-to-V

\[
\text{SD: (S (V \llangle L \gg)) ...} \] <= (NP)? (S (V ...))
\]

%Rule: Lowering-to-V

\[
\text{SC: #1 #2 #3 #4}
\]

\[
\text{EITHER RB: } \Rightarrow [\langle x_1: (V (V<\text{spine}>)) \rangle]
\]

\[
\text{OR LB: } \Rightarrow [\langle x_1: (V (V<\text{spine}>)) \rangle]
\]

The branching-directionality of the V-cluster is checked by means of the spine-feature: if the feature is on the leftmost V-daughter ({\langle x_1: (V (V<\text{spine}>)) \rangle}), the cluster is left-branching; otherwise ({\langle x_1: (V (V<\text{spine}>)) \rangle}) it is right-branching. Note that the latter expression also accounts for a non-branching V to comply with the default right-branching.

Again, both branching varieties must now apply in Dutch and the conditions must not apply in German. So, we replace them by parameters and add the relevant statements to the Dutch parameter file:

\[
\text{SC: #1 #2 #3 #4}
\]

\[
\text{EITHER RB: } \Rightarrow [\langle \text{SPINE-RIGHT} \rangle]
\]

\[
\text{OR LB: } \Rightarrow [\langle \text{SPINE-LEFT} \rangle]
\]

%Rule: Lowering-to-V

\[
[+] [RB]
\]

\[
[+] [LB]
\]

\[
\text{SPINE-RIGHT: } [\langle x_1: (V (V<\text{spine}>)) \rangle]
\]

\[
\text{SPINE-LEFT: } [\langle x_1: (V (V<\text{spine}>)) \rangle]
\]
For left-branching (68) and right-branching (70), the effect of the rule now is as follows:

```
S  |  S  
  V<rule<<<<<<<<<<<<<<<<<<<<<<<<<  V<rule<<<<<<<<<<<<
    |    
  V  |  V  
    |    
  V<spine>  |  V<spine>  
SIM | SIM dansen willen ik  |  SIM willen SIM dansen ik
```

Lowering-to-V, Tree-Pruning-1, Surface-Category-Assignment ==>

```
S  |  S  
  V  |  V  
    |    
  V<spine>  |  V<spine>  
    |    
  V<spine>  |  V<spine>  
SIM dansen willen SIM ik  |  SIM willen SIM dansen ik
```

The other two cyclic cluster-building rules, Participle and Copula Insertion, do not need to be adjusted to the Dutch L-verbs, because they never apply in the specific L-verb configuration. The postcyclic rules AUX-Delimitation, SIM-Deletion, AUX-Attraction, Affix Handling, V-Final, and Mince do need adaptation to the existence of partly left-branching clusters in Dutch. This should be done along the same lines as shown above for Lowering-to-V, i.e. by adding parametrized spine conditions to the left-branching variants of SD and/or SC as already implemented for German. We have not yet carried these adaptations through, however.

### 5.5 Other adjustments to the rules

#### 5.5.1 Argument Functions

Recall that in the English grammar we needed to cater for deviant argument functions listed in the lexicon (see section 4.2.4), because with some predicates the middle argument is direct object instead of indirect object. In Dutch and German, we come across another exceptional argument frame: in some cases, when there are only two arguments the second is indirect object instead of direct object. This occurs, for example, with the Dutch verbs *gehoorzamen* ("obey") and *toebehoren* ("belong"), and German verbs like *helfen* ("help"), *vertrauen* ("trust") or *gratulieren* ("congratulate"). The phenomenon shows up most visibly in German, which has different
overt case markings for direct and indirect objects (i.e. accusative and dative, respectively). This is illustrated in (75).

(75)  Ich helfe dir.
      I help you-DAT

(= I help you)

To cater for these lexical argument frames in the implementation, we extend the rule as follows:

Now, if the lexicon lists any deviant argument function

\{(?[@LEX]: x1): (af<IO>) \} or \{(?[@LEX]: x1): (af<DO>)\}, the relevant argument is assigned that function instead of the default one (x3 plus "<af<IO>>" or x2 plus "<af<DO>>"). This extension does not need to be parametrized, because the condition will simply never be fulfilled in English.

5.5.2 Perfective Auxiliary

Unlike English, languages like Dutch and German have two different auxiliaries for which the choice depends on the lexical verb: the auxiliary is either hebben/haben (have), or zijn/sein (be), respectively.\footnote{It is often assumed that the choice of auxiliary depends on some semantic property of the lexical verb. As Seuren (1996, p.220-221) argues, however, “it has so far proved impossible to present a general principle determining the correct perfective auxiliary choice” and therefore “for the moment we will [...] simply mark those verbs that (normally) take zijn (or its equivalent) in the lexicon”.} This is illustrated in examples (77) and (78).

(76)  a    John has waited.
      b    John has left.

(77)  a    Jan heeft gewacht.
      b    Jan is vertrokken.

(78)  a    Jan hat gewartet.
      b    Jan ist abgefahren.
We extend the rule Perfective-Auxiliary (see section 4.2.8) to accommodate for this choice:

%Rule: Perfective-Auxiliary
---> SD: (S ... (V{type<PERF>> "FREC" # } ...)  
       (V{type<LEX>> ...I)  
---> COND: (x2 = [@HAVE]) & 
         ({{?[@LEX]: x1}: (<BE>) -> (x2 = [@BE])})  
---> SC: #1 #2 ==> #1 &x2

Now, the rule looks up the lexical verb (V{type<LEX>>}) in the lexicon and if it is marked for the be-variant of the perfective auxiliary (<BE>), the terminal zijn or sein ([@BE]) is substituted for the abstract filler. Otherwise, the have-variant ([@HAVE]) is used. Again, there is no need for parametrization.

5.5.3 Participle

A complication with the Dutch and German grammars is the so-called Infinitivus pro Participio effect (or *Ersatzinfinitiv*). This effect is illustrated in (79): in verbal clusters in the perfect tense, the verb after the auxiliary does not surface as a participle (gewild) but as an infinitive (willen).

(79) a * Jan heeft Marie gewild kussen.  
b Jan heeft Marie willen kussen.

For the definition of the rule Participle in Dutch this means: “The rule is inoperative when the highest lexically filled non-spine V-node in the V-cluster has a spine brother node labelled ‘V’” (Seuren 1996, p.225-6).

Recall our original implementation of Participle and its parametrization for branching directionality (see sections 4.3.1 and 5.4):

%Rule: Participle
---> SD: (S # (V{rulf><1>}) ... ) == (S ... # (V{<spine> ==}) #  
---> COND: (fI features "<PAP>") & (x2 = "(AFT-EN)")) |  
        (fI features "<PRP>") & (x2 = "(AFT-ING)"))  
---> SC: #1 #2 #3  
       [EITHER RB] ==>  
       #1 #2<^rulf<PAP,PRP> {&x2 >> #3<spine>}<^type,<spine>  
       [OR LB] ==>  
       #1 #2<^rulf<PAP,PRP> {#3<spine> << &x2}<^type,<spine>

To account for the Dutch IPP effect, we adjust the rule as follows, with the appropriate settings in the Dutch parameter file:

---> SD: (S # (V{rulf><1>}) ... ) == (S ... # (V{<spine> ==}) #  
       ... 3 )  
---> SC: #1 #2 #3  
       [SOMETIMES PPI] ==> [@PPI] #1 #2<^rulf<PAP,PRP> > #3  
       [EITHER RB] ==> [@NOT-PPI]  
       #1 #2<^rulf<PAP,PRP> {&x2 >> #3<spine>}<^type,<spine>
%Rule: Participle
[+][PPI]
PPI: [\{x3: (V<spine> ...) \}]
NOT-PPI: [\{^\{x3: (V<spine> ...) \}\}]

Now, if the PPI-condition (\{x3: (V<spine> ...) \}) is fulfilled, no participle affix is adopted by the lower V-node, but the inducing rule feature is removed anyhow (\#2<^\{rulf<PAP,PRP>\}>). Otherwise, the normal right-branching structural change applies. In English, the condition parameter [NOT-PPI] remains empty.

There is one particular verb in Dutch with which the Infinitivus pro Participio effect has to be prevented from taking place: the verb *zijn* in constructions as given in (80a); see also Seuren (1996. p.255-257).

(80) a ... dat dit niet te doen is geweest / geweest is.
    ... that this not to do is been.

    b * ... dat dit niet is wesen te doen.
    ... that this not is be to do

("... that it has not been humanly possible to do this.")

The Infinitivus pro Participio condition would lead to the ungrammatical (80b). Seuren suggests to solve this by giving the infinitive cluster (te doen in the example sentences) the surface category of adjective, so the PPI-condition will not be fulfilled. In the implementation, we realize this by adjusting the assignment of the SCA feature in the rule Predicate Raising (see section 5.2.1):

%Rule: Predicate Raising:
---> SD: (S # (V<rulf<PR>1> ...) # <=2 #
    (S<^\{type<FIN>\} # (V ...))3 # ==> \{ \} #)
---> COND: (\{^\{x2 is_empty\} \} ==> x2 plus "^\{af\}"
    (f4= "<SCA>") &
    (f1 features "<PR<SCAadj>>" \} (f4 = "<SCA<ADJ>>")
---> SC: #1 #2 #3 #4 #6 #7
    ==> #1 #2<rulf<PR> \} #5<\{&f4,spine\} &x2 #6

Now, if the Predicate Raising verb has a special marking on its rule feature ((V<rulf<PR>1> ...) with f1 features "<PR<SCAadj>>"), this marking is copied to the SCA-triggering feature (f4 = "<SCA<ADJ>>"). This adjustment does not need to be parametrized.

In German, the Infinitivus pro Participio effect is closely entwined with the R-verbs (see section 5.4.2). As such, it is not catered for in the implementation yet. See Seuren (1996, p.271) and Richter (2000) for the details.

---

82 In fact, Seuren (1996) gives a second reason for assigning the cluster te doen the category of adjective. In the original definition of End-Cluster-Arrangement, adjectives were also to climb up through the cluster, so this accounted for the cluster-initial position of te doen. However, the revised version of the rule (as given by Seuren 1997 and as implemented by us, see section 5.2.4) does not apply to adjectives so what is generated at present is ...dat dit niet *is* geweest / geweest is te doen instead.
5.5.4 Lowering

The third variety of Lowering. Lowering to the right, is sensitive to a number of conditions as were discussed above for English (see section 4.3.2). In Dutch and German, these conditions are even more complex. For both languages they are defined as follows:

"L_right does not cross an embedded S or /S, nor an adverbial to the right of V. An Adv2 not marked H and lowered by L_right may stop before NP[DO]." (Seuren 1996, p.225/283)

The first part of this definition can be integrated into the implemented rule by means of the following parametrization:

%Rule: Lowering-to-the-Right

$$\rightarrow$$ SD: (S # (V<rulf<L>R>>1 ...)) # (NP ...)? {(NP )}? (S =>) # ( ...)) #)

$$\rightarrow$$ SC: #1 #2 #3 #4 #5

$$\Rightarrow$$ [(x1: {NP })? (S<type> ) | {x1: {(NP )}? (S<SLASH> )} | (x1: (PF )} [@STOP-BEFORE]]

#1 #3 #2<SCA,"rulf<L>> #4 #5

$$\Rightarrow$$ [(x1: {NP })? (S<type> ) [{NOT-CROSS}]]

#1 #3 #4 #2<SCA,"rulf<L>> #5

%Rule: Lowering-to-the-Right:

STOP-BEFORE: [[ {x1: (ADV )}]

NOT-CROSS: [{x1: {NP })? (S<SLASH> ) | {x1: (PF )} | {x1: (ADV )}]

With the given values for STOP-BEFORE and NOT-CROSS in Dutch and German, the rule always has one output structure: if the argument-S contains an embedded S or /S ((x1: {NP })? (S<type> ) | {x1: {(NP )}? (S<SLASH> )} or an adverbial ((x1: (PF ) | (x1: (ADV ) the lowered element is inserted before it; otherwise, the element is inserted at the far right. In English, both parameters remain empty.

The second part of Seuren’s definition is more complicated and amounts to the following adaptation of the rule and parameter values for Dutch and German:

$$\rightarrow$$ SD: (S # (V<rulf<L>R>>1 ...2)) # (NP ...)? {(NP )}? (S =>) # ( ...)) #)

$$\rightarrow$$ SC: #1 #2 #3 #4 #5

$$\Rightarrow$$ [(x3: {NP })? (S<type> ) | {x3: {(NP )}? (S<SLASH> )} | (x3: (PF )} [@STOP-BEFORE]]

#1 #3 #2<SCA,"rulf<L>> #4 #5

$$\Rightarrow$$ [(x3: {NP })? (S<type> ) [{NOT-CROSS}]]

#1 #3 #4 #2<SCA,"rulf<L>> #5

%Rule: Lowering-to-the-Right:

STOP-BEFORE: [[ {x3: (ADV )}]

(f1 features "<type<ADV2>>" &

\{[@LEX]: x2: (GSTOP) \} & \{x3: (NP<af<DO>>) \})

NOT-CROSS: [{x3: {NP })? (S<SLASH> ) | {x3: (PF )} | {x3: (ADV )}]

Now, if the lowered element is a level 2 adverbial (f1 features "<type<ADV2>>") and is not specially marked in the lexicon (\{[@LEX]:
x2); \{ <\text{<NOSTOP>} > \}; the feature <\text{<NOSTOP>} > is equivalent to Seuren’s symbol H), it may halt before a direct object NP (\{ \text{x3:} \ (\text{NP<af<DO>} > ) \}). Note that only in this case the rule outputs two structures instead of just one.

### 5.5.5 Particle-Insertion

Recall that in English, the rule Particle-Insertion is responsible for inserting the particle \text{to} before infinitives, unless the higher predicate is marked as [-to] in the lexicon (see section 4.3.6). The Dutch and German variants of Particle-Insertion function differently. Seuren’s definition for the Dutch rule is:

"Tr-INSERTION is a corollary of any cyclic S inducing SD or PR unless barred by [-te]. The highest lexically filled V-node, except \text{v[0]}, in the V-cluster of the argument-S that has undergone SD or PR left-adopts \text{p_r[te]}. If there are two highest lexically filled V-nodes, the non-spine V left-adopts \text{p_r[te]}."

(Seuren 1996, p.226)

And for German:

"Zu-INSERTION is a corollary of any cycle inducing [+zu] or SD without PR: The highest lexically filled V-node, except \text{v[0]}, in the V-cluster of the argument-S left-adopts \text{p_r[zu]}. If there are two lexically filled V-nodes of equal height in the cluster, not counting \text{v[0]}, the non-spine V left-adopts \text{p_r[te]}."

(ibid., p.283)

Our preliminary implementation of this is first to have the rule Predicate-Raising introduce a triggering feature in Dutch and a blocking feature in German:

%Rule: Predicate-Raising:

\[
\begin{align*}
\text{SD: } & (S \ # \ (V<\text{rulf<PR}>1) \ \ldots ) \ # \ <==2 \ # \\
& (S<\text{<type<FIN>}> \ # \ (V \ \ldots )3 \ # \ ==> \ # ) \ # \\
\text{COND: } & (\text{^xAf is empty} \ => \ x2 \ plus \ "<\text{af}>>\") \ & \\
& (f4 = \ "<\text{SCA}>>\") \ & \\
& (f1 \ features \ "<\text{PR<SCAdj>>}" \ => \ (f4 = \ "<\text{SCA<ADJ>>}")) \\
\text{SC: } & \#1 \ #2 \ #3 \ #4 \ #5 \ #6 \ #7 \\
& \Rightarrow \ [f1 \ features \ "<\text{OPT}>>\]) \ #1 \ #2<\text{rulf<PR>>} \ #3 \ #4 \ #5 \ #6 \ #7 \\
\text{EITHER BB} \Rightarrow > \\
& \#1 \ #2<\text{[@PRT]<rulf<PR>>} \ << \ #5<\text{&f4, spine}> > \ }[@\text{REM-PRT}] \ & x2 \ #6 \\
\text{OR LB} \Rightarrow > [\text{[L-COND} \\
& \#1 \ #5<\text{&f4, spine}> => \ #2<\text{[@PRT]<rulf<PR, LB>>}][@\text{REM-PRT}] \ & x2 \ #6 \\
\end{align*}
\]

**Dutch:**

%Rule: Predicate-Raising:

[@PRT]: PRT,
[@REM-PRT]: <^PRT>

**German:**

%Rule: Predicate-Raising:

[@PRT]: NOPRT,
[@REM-PRT]: <^NOPRT>

The rule of Particle-Insertion itself becomes heavily parametrized to cover both the English default insertion (DEF) and the Dutch/German triggered insertion (TRIG):
%Rule: Particle-Insertion
--> SD: (S ... #
  [EITHER DEF] (V ==1) <= (S<SLASH> <= # {(V )? (V ==1) {==}?) #
    =>)2
  [OR TRIG] (V[@PRT] ==1) ... # (V ==2) #
  --> COND: (x3 = "(Prt-[TO])") & ^{(?[@LEX]: x1): (V<-[TO])} &
  [EITHER DEF] ^{(x2: ... (AFF "PRES")} | (x2: ... (AFF "PAST")}
  [OR TRIG] ^{(x2: "SIM")}
  --> SC: #1 #2 #3
  [EITHER DEF] == #1 #2 (x3 >> #2<spine>)<spine>
  == [([?[@LEX]: x1]: (V<to>)) #1 #2 #3
  [OR TRIG] == #1 #2[@REM-TO] { x3 >> #3[@SPINE] }[@REM-SPINE]

In German it is even more complex than this. There we need an additional rule for the insertion of zu triggered by the lexical feature [+zu]:

%Rule: Particle-Insertion-2
--> SD: (S (V ... # (V ==1) # ... (V ==2))
  --> COND: ^{(x1: "0") & ^{(?[@LEX]: x2): (V<zu>)} &
    (x3 = "(Prt-zu)")
  --> SC: #1 #2 #3
    => #1 x3 >> #2

5.5.6 SIM-Deletion

Recall that in English, the postcyclic rule SIM-Deletion removes the simultaneous tense operator if followed directly by be (see section 4.4.2). In Dutch and German, on the other hand, the rule is almost trivial: the tense operator SIM is always removed, no matter what node follows it. Thus, to cover also the right-branching V-clusters of Dutch, we parametrize the SD:

%Rule: SIM-Deletion
--> SD: (S ... (V # (V<1 "SIM") # (V )? # [@FOLLOW] #
  --> COND: None
  --> SC: #1 #2 #3 #4 == #1 #3 #4<&f1>

English:
%Rule: SIM-Deletion
@FOLLOW: (V &BE)

Dutch:
%Rule: SIM-Deletion
@FOLLOW: ( ... )

Now, in Dutch it does not matter what node follows the tense operator ([@FOLLOW] with @FOLLOW: ( ... )) for SIM-Deletion to apply.

For the left-branching clusters of German, we need to use branching parametrization in the SD and SC again (see also section 5.4.2):
5.6 Conclusion

To conclude this final chapter on our generator, let me summarize what we claim to have contributed to the model of Semantic Syntax by implementing the grammars for Dutch and German the way we have done. Both of our research questions as formulated in Chapter 1 are relevant here. With respect to the descriptive adequacy of Semantic Syntax, we have tried to establish whether Seuren’s grammars for Dutch and German are internally consistent and account for the sentences that they are claimed to account for. With respect to the model’s crosslinguistic adequacy, we have tried to reach a higher level of generalization by constructing a language-independent grammar with parameters that can be set differently for each of the three languages under consideration.

5.6.1 Descriptive adequacy

As for English, I think we may again conclude that by and large we have succeeded in implementing the Semantic Syntax transformational grammars for Dutch and German satisfactorily. The implemented grammars remain very close to Seuren’s formalizations and generate the surface structures of almost all of the sentences that Semantic Syntax is claimed to cover so far.

Again, however, there are a few constructions that the implementation does not cover:

- The main phenomenon that has been left unaccounted for so far, are the German R-verbs (see section 5.4.2). They require similar adaptations to the rules as we have made for the Dutch L-verbs, but we have not carried these through. A logical consequence of this is that the Infinitivus pro Participio effect (see section 5.5.3), which is closely entwined with the R-verbs, has also not been implemented for German yet.

- Seuren’s (1997) account of the Dutch L-verbs on the other hand has been implemented (see section 5.4.2). The only thing we have not worked out in this respect are the small adaptations needed in a few postcyclic rules to the existence of partly left-branching clusters in Dutch too.
• Another aspect of the Dutch implementation that is not fully satisfactory is the treatment of te+infinitive clusters that are considered to be adjectives. At present, these clusters are not affected by End-Cluster-Arrangement, although they should be to end up in the right position (see footnote 82, p.177).

• Finally, the insertion of te in Dutch and zu in German has only been implemented in a provisional way (see section 5.5.5). In our view, the exact conditions on this insertion have to be further examined before a satisfying implementation can be accomplished.

Aside from these points, the implementation generates all of the Dutch and German sentences that Seuren claims the Semantic Syntax model covers. There are, however, also some differences between Seuren’s formalization and our implementation in how these sentences are accounted for:

• Downgrading 2 is not implemented as a separate rule, but rather integrated into Predicate Raising (see section 5.2.1), just like Downgrading 1 has been integrated into Subject Raising and Subject Deletion (see sections 4.3.5 and 4.3.7).

• The re-assignment of argument functions associated with Predicate Raising is not integrated into this rule, but instead it is taken care of by re-calling the rule Argument Functions, that is also part of the Pre-cycle (see again section 5.2.1).

• In the implementation, we need an extra rule within the postcyclic grammar Affix Handling to take care of compound verbs containing a particle (see section 5.2.2). This rule, called Particle-Participle, has no specific counterpart in Seuren’s formalization.

• For the implementation of End Cluster Arrangement, we have used Seuren’s (1997) revised version instead of the original definition by Seuren (1996). The revised version is much simpler and fits in naturally with the GRAMTSY machinery, whereas the original rule involved the assignment of weights and as such would have been a lot more difficult to implement. We have reached an additional simplification of the rule by establishing that the condition on participles climbing over particles is more general than Seuren (1997) assumes (see section 5.2.4).

• To account for the Dutch L-verbs, we have added a rule to the Pre-cycle called Directionality-Switch, which has no counterpart in Seuren’s formalization (see section 5.4.2). Like the other precyclic rules, it does not involve a real transformation, but only the insertion of a feature to mark possible L-verb configurations as such.

5.6.2 Crosslinguistic adequacy

The main conclusion of this chapter with respect to the crosslinguistic question, in our view, is that the three transformational grammars of English, Dutch and German
have proven to be very well suitable for parametrization. The means of our programming tool GRAMTSY to parametrize both grammars and rules have been powerful enough to collapse the three grammars into one general grammar, with relatively small parameter setting files for each separate language.

Of over sixty grammar and rule calls in total, only nine are not shared by all three languages (see also Table 5.1 and footnote 67, p.144): four rules are shared by Dutch and German, and five are specific to just one language. These parametrizations are not completely independent of each other. For example, the fact that Argument Functions is called only in Dutch and German is fully determined by the fact that Predicate Raising is called only in these two languages (see section 5.2.1). Unfortunately, GRAMTSY does not offer a straightforward means to link these parametrizations – both rules simply apply [OFTEN]. The only way to express that the occurrence of one rule depends on the occurrence of another is to group them into a subgrammar and parametrize the call of this subgrammar. This is what we did with End Cluster Arrangement, a rule of which the occurrence seems to rely (partly) on the occurrence of V-Final: we placed the call for End Cluster Arrangement in the subgrammar of V-Final (see section 5.2.4), and parametrized the subgrammar call. Because of rule orderings, however, this solution may not always be possible. Some further study of the interrelation between the parameters is in order, for example, to determine whether the occurrence of V-Final depends on the occurrence of Predicate Raising.

Besides this grammar parametrization to account for which grammars and rules are to be applied in a language, we have also made extensive use of rule parametrization. Thus, we have parametrized all rules that involve lexicon lookup or a direct reference to lexical fillers (see sections 5.3.1 and 5.3.2). By treating the lexical parameters as global rather than rule-specific parameters, we have managed to keep their number limited to six (see Table 5.3). This also involved regarding the parameters for the copular verb and the perfective auxiliary as instances of the same parameter.

Differences in branching directionality have also been implemented by means of rule parametrization. For the left-branching clusters of German, we have parametrized four cyclic rules and six postcyclic rules (see section 5.4.1). The cyclic rules are the ones involved in building up the V-cluster, so only their SC-parts had to be parametrized. These are not just Lowering-to-V and Predicate Raising, as mentioned by Seuren, but also Participle and Copula Insertion (see footnote 79, p.160). The postcyclic rules are the ones that need to recognize the clusters and keep their directionality intact, so this involved more complex parametrization of all rule parts. The main reason for this complexity is that GRAMTSY is essentially a string-based system with no means to abstract away from the linear ordering in SD and SC. As such, it does not offer a straightforward mechanism to implement concepts like Seuren’s “α-adopt” and “spine”, where α is left in English and Dutch but right in German, or where the spine is to the right in English and Dutch but to the left in German. For example, it is not easy to express something in GRAMTSY like: “if A and B are
brothers and B is on the spine, then A is adopted by B” if the order of A and B can vary.

Finally, let us take a look at a number of more isolated details where we have brought the rules for English, Dutch and German more in line with each other:

- Seuren (1996) assumes the particle of compound verbs to be on the right-hand side in Dutch and on the left-hand side in German. In the implementation, however, we have followed Seuren (1997) in taking the particle to be always on the left-hand side (see footnote 73, p.148).

- We assume the abstract adverbial EMPH to exist not only in English but also in Dutch and German, although Seuren (1996) has not included it in his formalization (see footnote 77, p.156).

- Since Dutch and German do not have a progressive form, the rule Present Participle is not part of Seuren’s grammars for these languages. In the implementation for English this rule has been combined with Past Participle into one general rule Participle (see section 4.3.1). The absence of present participles in the Dutch and German rule systems is no reason to adjust the implemented rule Participle for these languages, however, because the triggering rule feature simply never occurs (see footnote 80, p.160).

- We have argued that Seuren (1996) is right in assuming that German has the rule Affix Handling too, just like Dutch and German do, even though there is no need to change the order of verb and affix (see section 5.4.1).

- We have shown that the Dutch L-verbs are highly similar to the German R-verbs (see also Table 5.5, p.169), although the conditions on their directionality switch differ slightly. Because of this directionality switch, the Dutch and German grammars differ from the English one in that they need both left-branching and right-branching variants of rules that affect the V-cluster (see section 5.4.2).

- We have tried integrating the rules To-Insertion (English), Te Insertion (Dutch) and Zu-Insertion (German) into one rule Particle-Insertion, but this proved to be very difficult (see section 5.5.5). Our tentative implementation of this rule is highly parametrized because the three rules differ so much. Further study of the conditions on and effects of this operation is needed to determine whether it is indeed essentially the same rule in the three languages.
CHAPTER 6  

Semantic Syntax parsing

6.1 Introduction

In the previous chapters, I have shown that the model of Semantic Syntax is well suited to be implemented as a generator. The main results of our technolinguistic undertaking are an improved notation, an explicit account of the differences and similarities within the grammars of three related languages, and a better understanding of the relation between formalization and theory (see sections 3.4, 4.5 and 5.6). We have located some problems, gaps and inconsistencies within the formalization and linked them to the underlying theory. In a number of cases, the process of implementing has required us to explicitly formulate theoretical principles behind the formalization. Thus, we have answered our research questions satisfactorily.

However, there is another aspect of the relation between theory and formalization that we have not scrutinized yet: the question whether Seuren's formalization is a necessary (or natural) consequence of the theory. In other words: could Semantic Syntax be formalized differently? Are all the transformational rules as defined by Seuren inextricably bound up with his type of analysis, or are some of the rules more or less a coincidental result of particular choices? These are the questions that we will examine in the present chapter.

In particular, we will investigate whether the generative character of Seuren's formalization is essential to the Semantic Syntax model. Following technolinguistic practice again, we will now investigate how a parser (rather than a generator) could be implemented. Seuren has the following view on Semantic Syntax parsing:

“The parsing procedure will presuppose the generative system but follow its own principles. It will use certain formal features of surface sentences, such as tense markings and categorial word labels, to set up chunks of SA on the basis of information from the Lexicon and the generative rules. A generative re-run of the reconstructed SA will test the parsing result if necessary.”

(Seuren 1996, p.23)

However, instead of using the generative rules to set up SA-chunks and re-run the reconstructed SA as Seuren suggests, we will examine to what extent the generation process can genuinely be inverted. That is to say, we will demonstrate how to implement a transformational process from Surface Structure to Semantic Analysis in much the same way as it was implemented the other way around. Note, however, that the focus of this inversion is the process as a whole, rather than the individual generative rules. There are two reasons for this. First, simply inverting every single
rule would not help much in answering the question as to their necessity status. And second, it might well be impossible to invert every rule, as it is a well-known property of transformational grammars that they are not guaranteed to be reversible.

It is important to stress that we consider the implementation of a parser as a mere exercise, which only serves to shed some extra light on the questions mentioned above. It is not our aim to actually implement a fully fledged parser, as we have done with the generator. Instead, we will only demonstrate how to implement a basic parser. Based on this, we will draw our conclusions as to what can be expected when one sets out to extend this basic implementation into a full parser. We will indicate which problems remain open for future research and hint at possible solutions.

A standard difficulty in parsing is the problem of combinatorial explosion, caused by a multitude of ambiguities encountered. It is not to be expected that our exercise in Semantic Syntax parsing will have anything to offer to solve this principled problem. At best, we hope to show that Semantic Syntax parsing is compatible with other approaches, in the sense that standard strategies (like underspecification or the use of probabilities) can be applied in our account as well.

Since we have already looked at the language-specific elements versus the more universal aspects in the previous chapters, it will now suffice to look at only one language. We will restrict ourselves to Dutch. As this is also one of the languages studied in the previous chapters, we will be able to compare the parser to the generator. Also, we can benefit from a large expertise on Dutch parsing present within our research group. We will be able to use a front-end surface parser called AMAZON, developed at the University of Nijmegen. Thus, we join a long tradition of parsing research carried out here. An advantageous side effect of this choice is that our findings will be (re-)usable for other AMAZON-related research.

The Semantic Syntax parser for Dutch that we envisage is a two-stage system, as depicted in Figure 6.1. First, we will have the AMAZON module produce a Surface Structure. This structure will be transformed by a second module into the corresponding SA according to the model of Semantic Syntax. In the following two sections, I will describe the AMAZON module and our basic implementation of the Semantic Syntax module.
6.2 The surface parser AMAZON

AMAZON is a structuralistic surface parser for Dutch, developed at the University of Nijmegen over the past three decades (see van Bakel 1975, van Bakel 1984, Coppen 1995, van Dreumel & Potjer 1998). The Surface Structures produced by AMAZON are not identical to the Surface Structures generated in Semantic Syntax. Since AMAZON has been developed in the same technolinguistic tradition, however, it will turn out to be re-usable – and in fact very useful – as a pre-module to our semantic parser.

AMAZON produces structuralistic surface structures for Dutch sentences, i.e. each sentence is divided into constituents and subconstituents. These consituents are, however, not assigned their grammatical functions or thematic roles by this surface parser. The analysis is based on structuralist grammars, like Haeseryn et al. (1997).

For example, AMAZON assigns the following structure to the sentence in (81):

(81) Ik zag dat Jan lachte.
I saw that John laughed.

A separate post-module called CASUS is designed to do that. For our purposes, we replace CASUS by our own semantic parser, as described in the next section.
In the Amazon analysis, every sentence (SE) consists of five parts:

- TOP (at most one topicalized constituent, either an XP or an embedded clause);
- V (finite verb, in main clauses), or C (complementizer, in embedded clauses);
- MI (middle field, any XPs in between V/C and CL);
- CL (verbal cluster);
- EX (any extraposed elements, XPs and/or embedded clauses).

In main sentences, the second constituent (V) is always filled, since Dutch obligatorily has verb-second in main clauses. One or more of the other four constituents may remain empty: for example, in the tree structure above, the MI and CL nodes are empty nodes. In embedded clauses, on the other hand, the CL constituent is always filled. The C, MI and EX nodes may remain empty, and the TOP node may be absent.

Besides a categorial label for each node, Amazon also delivers a large amount of additional information by means of features attached to each label. So, for example, the determiner "ik" above has the following feature bundle attached to it:

(np, -clit, ea, -wh, +def, first, -plu, -neu, +cnt)

These features mean that this particular determiner is (part of) a regular NP, not enclitic, an external argument (because of its nominative form), not a wh-element, definite, first person singular, masculine or feminine, and a count noun.\(^{84}\)

In case of an ambiguity, Amazon usually either uses underspecification or it chooses the most probable analysis. In rare cases, it yields more than one analysis for a particular sentence.

---

\(^{84}\) See also Van Dreumel’s website (http://lands.let.kun.nl/TSpubic/dreumel/amazon_results_Seur_en.html), which contains all of Amazon’s Surface Structures for the Dutch example sentences in Seuren (1996).
6.3 A basic semantic parser

6.3.1 From SS to SA

Our own first step is to convert the AMAZON output to a simpler and flatter structure, which only contains the nodes that have Semantic Syntax counterparts. Thus, we have a simple cosmetic routine (that comes with AMAZON) remove all nodes except SE, NP, V, CL, and C; the SE and CL nodes are relabelled as S and V, respectively. We also erase all the featural information AMAZON delivers. The example tree now looks as follows.\(^{85}\)

![Tree Diagram]

This structure will be the input to our semantic parser. We do not consider it to be a problem that we do not start from a "standard" Semantic Syntax surface structure (SS). In our view, the exact structural make-up of the Semantic Syntax SS does not seem to be an essential part of the theory, but rather a somewhat peripheral aspect of the formalization. Thus, our parser simply has a somewhat different input compared to the output of the generator, but as already explained above, we will not be aiming to invert all the individual generative rules anyhow (see section 6.1).

The task of the parser is to get from SS to SA through a series of transformations. The target SA of our example sentence is the following, which corresponds to the SAs given by Seuren (1996):

\(^{85}\) The empty V-node after "zag" and the double V-node over "lachte" result from the relabelling of CL as V. These nodes will be removed by the cosmetic rules Remove-empty-nodes and Tree-Pruning (see below).
For the implementation of the basic semantic parser we will use the tool TREMA (van Bakel & Boon 1998), a transformational system that allows one to specify transformations in much the same way as they are usually written down in theoretical linguistics. This functionality makes TREMA the perfect tool for technolinguistic implementations. It is in fact a newer and faster version of GRAMTSY, which we have used for our generator.

Like GRAMTSY, TREMA applies transformational rules to a given input tree. And, as within GRAMTSY, each rule consists of a structural description (SD), a set of conditions (Cond), and a structural change (SC). The SC contains one or more basic operations to be performed on the tree if the conditions are met. In TREMA, the basic tree operations and conditions are predefined. The operations consist in the adding, removing, moving and/or copying of (sub)trees, features or labels.

An example of a somewhat trivial rule in our parser is:

%Rule Tree-pruning
--> SD: (S ... ( ( ... )1)2 ... )
--> Cond: [label_eq &1 &2]
--> SC: [DeleteLevel &2]

This is a cosmetic rule, which removes a non-branching node whose daughter has the same label as she has herself (in our example tree, this rule removes the highest of the two V nodes over "lachte"). A similar example of a cosmetic rule is the following, which removes all empty nodes:

%Rule Remove-empty-nodes
--> SD: (S ... ( )1 ... )
--> SC: [Remove &1]

The result of these two rules on our example tree is straightforward:

```
S
  NP V V | S
  | | | C NP V
  | | | | | V
  | | | | | | | "ik" "zag"
  | | | | | | | | | "dat" "Jan"
  | | | | | | | | | | | "lachte"
```
These and a few other cosmetic rules are applied to the input structure as a whole. After that, the main rules will be applied cyclically and top-down. That is, each S-node with its direct daughters constitutes a cyclic domain in which all the rules are applied before proceeding to the next (lower) S-node. Thus, our little example tree is divided into two consecutive cycles:

```
  S
  |  |  |
  |  |  |
  NP  V  S  C  NP  V
  "ik" "zag" ...  "dat" "Jan" "lachte"
```

*first cycle  second cycle*

A crucial feature of Seuren’s generative approach is cyclicity. The core rules of Seuren’s grammars are cyclic rules, applying bottom-up to each successive S-cycle. They consist mostly in the raising or lowering of elements to a higher or lower S-subtree. As a side-effect, many of the S-levels disappear: when elements are moved out of an S-subtree, the subtree disappears. The overall effect can be seen in the two trees on p.189; the SS has only two S-levels, whereas the corresponding SA has six. TREMA supports cyclicity and allows the user to specify the desired directionality. Hence, there is no obstacle for us to implement top-down, cyclic transformations that are to mirror Seuren’s bottom-up, cyclic transformations. For example, we will have a rule Remove Complementizer mirroring the rule Complementizer Insertion (see section 4.3.11). The parser rule will remove the complementizer within the second cycle of our example tree.

We will have to supply the system with a number of rules with no generative counterparts in Seuren’s grammar, e.g. morphological look-up rules. The question is whether to process these rules globally or cyclically, i.e. in an early or a late phase. As a general principle, we choose to apply all rules as late as possible. This is conceptually more elegant (why look up a feature, if it is used only much later?) and often also more economical (if a tree is filtered out for some reason, it would be a waste if the system had already looked up all kinds of features for it). The downside is that applying these rules cyclically also means that they have to be tested on every single S-cycle.

In the remainder of this section, I will not discuss the implemented rules one by one, as I did in part I of this thesis for the generator. Instead, I will focus on a few main topics of Dutch syntax that Seuren (1996) treats in his grammar. We will implement these into a basic parser and will leave the remaining topics for further study (see section 6.4).

### 6.3.2 The tense system

Recall Seuren’s (1996) treatment of the tenses. As I explained in section 3.2.1, Seuren adopts a Reichenbachian tense analysis in which two tense operators are responsible for the present, past and perfect tenses. Thus, in our parser, we need a set
of rules that unravel the present, past and perfect verb forms into the verb stem plus the correct tense operators.

To recognize the various verb forms, we implement a rule that looks up each verb form in a word form lexicon. An inflected verb form is then replaced by two separate nodes: the appropriate affix plus the verb stem:

%Rule Verb-morphology
--> SD: (S ... (V ==1)2 ...) 
--> Cond: [count [lexicon MORPH &1]] > 0 
--> SC: [LeftAdjunction of [lexicon MORPH &1] to &2] & [Remove &2]

An entry within the word form lexicon looks as follows:

zag (V-(AFF="PAST") (V="zien"))

Thus, in the first cycle of our example tree, the V-node is replaced by a V-cluster:

\[
\begin{array}{ccc}
NP & V & S \\
& & \\
"ik" "zag" & ... \\
\end{array} \rightarrow \begin{array}{ccc}
NP & V & S \\
& & \\
& AFF & V \\
& & "ik" "PAST" "zien" \\
\end{array}
\]

Note that we choose to have the affix precede its verb, even though Seuren (1996) assumes an intermediate stage in the generation process where the verb precedes the affix. This stage is achieved by means of a rule called Affix Handling, which moves the affix from the left hand side to the right hand side of the verb. Apparently, Seuren assumes a morphological module that can only attach suffixes to their hosts if they are already in the right linear order. This assumption, however, does not follow from any necessary theorem within Semantic Syntax. We therefore conclude that Seuren’s generative rule of Affix Handling is not essential to the model, and thus can be dispensed with in our implementation.

Also note that we could perhaps make use of Amazon’s features (see section 6.2) for splitting up a verb form into a verb stem plus its affix. This may even eliminate the need for a separate word form lexicon, but we will not investigate this possibility any further here.

One of the main tense rules in our basic parser is the following:

%Rule Finite-Verb
--> SD: (S<FIN> ... (V (AFF ==1)2 (V ==))3 ...)4 
--> Cond: Match &1 with [&FIN] 
--> SC: [Remove &2] 
& [DeleteLevel &3] 
& [SetFeatures &4 to <PERF>] 
& [SetRoot (S<FIN> (V<FIN> &1) !4)]

This rule is responsible for raising the finite affix to become a tense operator. As such, it is the inverse of Seuren’s generative rule Lowering-to-V. It applies in any finite clause that contains a V-cluster with a finite affix. The affix node is removed,
along with its mother node. The S-node now becomes a perfective S, and the finite tense operator is attached to it by left-adjunction. The effect of this rule is as follows:

As can be seen, the output of the rule is one step closer to the target SA, shown on p.189.

Note that this type of cyclic rules constitutes a possible problem. In generating new S-levels – mirroring the disappearance of those levels as described above – the system also generates new S-cycles. Thus, the output of Finite-Verb contains an extra S-level compared to its input tree. In general, when a new cycle X is generated above the active cycle Y, cycle X instantly becomes the new active cycle. After processing X, the (top-down) system proceeds to the next lower cycle, which is Y again. This means that many cycles will be processed more than once. Bearing this in mind, we will need to formulate all the cyclic rules in such a way that they never apply twice. If we do so, there is no real problem, although the multiple processing of one particular cycle is of course not very elegant or economical.

Another, similar rule raises the perfective auxiliary to become the second tense operator:

```plaintext
%Rule Perfective
==> SD: (S<PERF> <= (V (V ==1)2 (V ==)>))3 ==>4
==> Cond: Match &1 with {&PERF}
==> SC: [Remove &2]
    & & [DeleteLevel &3]
    & & [SetFeatures &4 to <LEX,PaP>]
    & & [SetRoot (S<PERF> (V<PERF> "PREC") !4)]
```

This rule does the same thing as Finite-Verb above, except that it looks for a perfective auxiliary instead of a finite affix. Note that, technically, we do not raise this auxiliary itself but rather the abstract filler "PREC" with which we replace it. Another difference is that the new node S<LEX> receives an additional feature PaP, which will trigger the rule Participle on the next cycle (see below). Again, as with Verb-Morphology, we might benefit from AMAZON’s features in order to detect the perfective auxiliary, but we will not pursue this option.

Our example sentence does not contain a perfective auxiliary – it is in the imperfective tense. According to Seuren, however, imperfective clauses contain a second tense operator just as well, albeit an empty operator (or, as we have argued in section 3.2.1, with the abstract filler SIM). Thus, we implement a rule Imperfective:
%Rule Imperfective

--> SD: (S<PERF> ==> (V<PERF> ==>) <=>)1
--> SC: [SetFeatures &1 to <LEX>]
    && [SetRoot (S<PERF> (V<PERF> "SIM") !1)]

The effect of this rule on the next cycle of our example tree is as follows:

```
NP  V  S<FIN>  ==>  V<PERF>  S<LEX>
"ik" "zien" ...
```

Note that we do not follow Seuren’s (1996) generative rule system closely in our parser with respect to the second tense operator. Seuren treats the perfective and imperfective on a par, in that they both have a tense operator that is lowered onto the V-cluster; it is only in the Postcycle that the empty tense operator is removed. We, on the other hand, have two separate cyclic rules: one for raising the perfective tense operator, and another for introducing the simultaneous tense operator directly at its higher SA-position. That is, in our parser the simultaneous tense operator is not raised from a lower position. As far as we can see, it would indeed be feasible to first introduce the simultaneous operator within the V-cluster and then have one rule raise the second tense operator, whether it be the perfective or the simultaneous one. But we will not examine this possibility any further.

Finally, we include a rule called Participle to remove the participle affix from past participle clusters:

%Rule Participle

--> SD: (S<PaP> <=> (V (AFF "PREC")1 ==>)2 ==>)3
--> SC: [Remove &1]
    && [DeleteLevel &2]
    && [RemoveFeatures <PaP> from &3]

Note that the participle affix (AFF "PREC") has been retrieved by the rule Verb-morphology in the same way as the past and present tense affixes. The triggering feature <PaP> has been added to the S-node by the rule Perfective.

To summarize, we have included one global rule and four cyclic rules to parse the basic verb tenses according to Semantic Syntax. Table 6.1 lists these rules and their counterparts as implemented in the generator (see sections 4.2.8, 4.3.1, 4.3.2, 4.4.2, 4.4.6, 5.5.2 to 5.5.4, and 5.5.6). It is important to realize, however, that the parser rules in the right-hand column are generally not the exact inverse of the generator rules in the left-hand column. Thus, both Finite-Verb and Perfective can be regarded as the inverse of Lowering-to-V, but our implementation of Perfective additionally incorporates the effects of the separate generative rule of Perfective-Auxiliary. The two versions of Participle can be regarded as each other’s opposites. The rule Imperfective combines elements of SIM-Deletion and Lowering-to-V. Finally, the rule Verb-Morphology implements a different process from Affix Handling, but the former does render a counterpart to the latter superfluous.
Table 6.1 *Tense-rules in the generator and in the parser.*

<table>
<thead>
<tr>
<th>Generator</th>
<th>Parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lowering-to-V (cyclic)</td>
<td>Finite-Verb (cyclic)</td>
</tr>
<tr>
<td>Perfective-Auxiliary (precyclic)</td>
<td>Perfective (cyclic)</td>
</tr>
<tr>
<td>Participle (cyclic)</td>
<td>Participle (cyclic)</td>
</tr>
<tr>
<td>SIM-Deletion (postcyclic)</td>
<td>Imperfective (cyclic)</td>
</tr>
<tr>
<td>Affix-Handling (postcyclic)</td>
<td>Verb-Morphology (global)</td>
</tr>
</tbody>
</table>

### 6.3.3 Argument structure

The way in which we will analyse argument structure is similar to the way this is done in the CASUS system (see Coppen 1995): we retrieve an empty argument frame from the lexicon and assign the argument functions to the NP’s and S’s present. The tree is filtered out if any argument function remains unassigned, or if any NP or S remains without a function.

First, we need a rule to retrieve the argument frame:

%Rule: Get-frame

\[
\text{SD: } \langle S<\text{LEX}> ... (V \text{---}1)2 ... \rangle 3
\]

\[
\text{Cond: } \langle \text{count} \ [\text{lexicon FRAME } &1]\rangle > 0
\]

\[
\text{SC: } \langle \text{AddLeft} \ [\text{lexicon FRAME } &1] \text{ to } &3 \rangle
\]

\[
\& \& \langle \text{AddLeft } &2 \text{ to } &3 \rangle
\]

\[
\& \& \langle \text{AddFeatures } <\text{LEX}> \text{ to } &2 \rangle
\]

This rule consults a frame lexicon for every lexical V and inserts the appropriate frame (at the left) into the S-tree. Additionally, the rule moves the lexical verb itself to the far left of the S tree as well.

Since many verbs allow multiple different argument structures, the retrieved frame often contains two or more alternatives. We include a rule to transform a tree with alternative frames into alternative trees:

%Rule: Expand-OR

\[
\text{SD: } \langle S<\text{LEX}> \text{ ---} (\text{FRAME } \text{---} \text{ (OR } \text{---} (\ldots)1 \text{---}2 \text{---} (\ldots)) \rangle
\]

\[
\text{Cond: } \langle \text{LeftAdjunction of } &1 \text{ to } &2 \rangle
\]

\[
\& \& \langle \text{Remove } &2 \rangle
\]

Thus, the first lexical predicate *zien* in our example structure is listed in the lexicon as allowing either an NP or a finite clause as its direct object. Therefore, after the rule Get-frame has retrieved the argument frame from the lexicon, the rule Expand-OR replaces the tree by two alternative trees:
From this point on, these two trees will be processed separately. At the end of the cycle, the first one will be filtered out because the direct object argument function cannot be assigned to an NP (see below).

To assign the argument functions to the appropriate arguments, we implement the following set of rules:

%Rule Assign-subject
  --> SD: (S<LEX> <= (FRAME (&ARG <af<SU>> )1 ==>)
        ( &ARG =>2)3 ==>)
  --> Cond: [label_eq &1 &3]  
  --> SC: [RemoveFeatures <opt> from &1]
        & [AddLeft &2 to &1]
        & [LeftAdjunction of &1 to &3]
        & [Remove &3]

%Rule Assign-objects
  --> SD: (S<LEX> <= (FRAME ==>) (&ARG &ObjFeature )1)
          ( <af<SU>> ==>) ==>) ( &ARG =>2)3 ( &ARG <af<DO>> ==>)?)
  --> Cond: [label_eq &1 &3]
        & (! Match (ARG &f3 ) with ((ARG &ObjFeature )))
  --> SC: [RemoveFeatures <opt> from &1]
        & [AddLeft &2 to &1]
        & [LeftAdjunction of &1 to &3]
        & [Remove &3]

These two rules operate in a similar way. First, Assign-subject looks up the leftmost argument in the tree. If this argument has the right label according to the argument frame (either NP or S), it is marked as the subject. This rule applies only once per cycle. Second, Assign-objects looks up the rightmost argument and marks it as the direct or indirect object, according to its position. This rule may apply twice.

In the main clause of our example structure, which contains two arguments, both rules apply once:
Note that we assume that the arguments are in their canonical order subject – indirect object – direct object. Obviously, this is a simplification of the facts. In the next section, I will describe how we intend to account for order variations due to topicalization and question formation. Other order variations will be left out of account in our basic implementation.

Recall from section 4.3.9 that in extrapolation structures a dummy subject *het* takes the regular position of a subject finite clause in sentences like (82).

(82) Het schijnt dat Jan lachte.
    It seems that John laughed.

In order to correctly analyse such structures we add the following rule:

```
%Rule: Extrapolation
--> SD: (S<LEX> <= (FRAME (S<af<SU>),FIN> ) ==> ) (NP "het")1
       (S ==> ) ==> )
--> SC: [Remove &1]
```

This rule applies before Assign-subject. It removes the dummy NP-subject in order for the subject function to be matchable to the S-argument.

When the argument functions have been assigned, the remaining frame should be empty (as in the example above) or contain only optional argument functions. In both cases, the frame is removed by a rule called Remove-empty-frame:

```
%Rule: Remove-empty-frame
--> SD: (S<LEX> <= (FRAME (<opt> )?)1 ==> )
--> SC: [Remove &1]
```

If the frame cannot be removed because it is not empty, the tree is filtered out by the following rule:

```
%Rule: Frame-left
--> SD: (S<LEX> <= (FRAME ==> ) ==>)
--> FILTER: non-empty frame left
```

A similar rule called Arguments-without-function will filter out the tree if it contains NP- or S-arguments that remain without a function.
To summarize, we have included six cyclic rules and two filters to parse the basic argument structures according to Semantic Syntax. Table 6.2 lists these rules and their counterparts as implemented in the generator (see sections 4.2.4, 4.3.9, and 5.5.1). Assign-Subject and Assign-Objects can be regarded as the inverse of Argument-Functions. The two Extraposition rules mirror each other closely. Additionally, we need three technical rules in the parser to retrieve the argument frame from a lexicon and to distribute its functions onto the arguments present. Obviously, the two filters have no counterpart within the generator either.

Table 6.2 Argument structure rules in the generator and in the parser.

<table>
<thead>
<tr>
<th>Generator</th>
<th>Parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument-Functions (precyclic)</td>
<td>Assign-Subject (cyclic)</td>
</tr>
<tr>
<td></td>
<td>Assign-Objects (cyclic)</td>
</tr>
<tr>
<td>Extraposition (cyclic)</td>
<td>Extraposition (cyclic)</td>
</tr>
<tr>
<td></td>
<td>Get-Frame (cyclic)</td>
</tr>
<tr>
<td></td>
<td>Expand-OR (cyclic)</td>
</tr>
<tr>
<td></td>
<td>Remove-empty-frame (cyclic)</td>
</tr>
<tr>
<td></td>
<td>Frame-left (cyclic filter)</td>
</tr>
<tr>
<td></td>
<td>Arguments-without-function (cyclic filter)</td>
</tr>
</tbody>
</table>

### 6.3.4 Topicalization and questions

Recall the Semantic Syntax analysis of topicalized elements and question formation: elements marked `<+FOC>` or `<+WH>`, respectively, are attracted by a sentence initial operator. If (in Dutch) the attracted element is not the subject of the clause, AUX too is attracted with the effect of having the finite verb always in second position (see Seuren 1996, p.227-230).\(^{86}\)

We can have our basic parser deal with fronted elements if we implement the following rule:

```
%Rule Fronting
--> SD: (S<LEX> <= (FRAME ==> (NP<af<DO>> )) (NP ==>)1
      <= ($\&\ARG^<FOCorWH> ==>) 2 ==>)
--> SC: [AddFeatures <FOCorWH> to $1]
      & & [RightAdjunction of $1 to $2]
```

\(^{86}\) As Seuren notes, the verb is not actually in the second position: the attraction operator can be preceded by a level 0 adverbial, in which case there are two elements before the verb. AMAZON analyzes these adverbials as pre-semantic elements and reserves the top-position for the fronted element. Thus, we can still assume that the first NP within an SE is either the subject or a fronted element. It may be worthwhile to investigate whether one might use Amazon's TOP-node in this rule, instead of throwing it away at an early stage (see section 6.3.1).
This rule applies before the assignment of argument functions, but after the argument frame has been retrieved from the lexicon. If both the frame and the S-tree itself contain more than one function, it is possible that the arguments are not in their canonical order of subject – indirect object – direct object due to topicalization. That is to say, the first argument may be a fronted object instead of the subject. Thus, we have this rule (which applies optionally and for every occurrence) reconstruct all possible argument orders.

Obviously, this rule can be refined in more than one way. In its present form, it introduces a number of ambiguities. These could be prevented by using more information that AMazon provides: first, we could take into account subject-verb agreement (to exclude NP’s of the wrong number or person to be analysed as subjects); second, we could use AMazon’s WH-feature to force the fronting rule to apply with wh-words. What also needs to be solved is the problem of fronted elements that are arguments to embedded verbs.

### 6.4 Conclusion

We conclude from our exploratory study into Semantic Syntax parsing that it indeed seems to be possible to invert the generative process from SA to SS into an analyzing process from SS to SA, without giving up its transformational character. It appears to be unnecessary to follow the non-transformational strategy that Seuren suggests of setting up SA chunks and re-running the reconstructed SA by means of the generative rules (see section 6.1). We suspect this strategy is prompted by the desire to keep close to the formalization rather than to the theory. The theory is that natural language grammar deals with (at least) two distinct linguistic tree structures, semantic structure and surface structure, and that these two are linked to each other by transformational rules. Seuren has given this a generative formalization and apparently wants to adhere to this for the parsing procedure as well, but we have now made a reasonable case that this is the wrong approach from a technolinguistic point of view. What is essential to the theory is its transformational character, not its generative character. It can be formalized both in a generative and analytic way, and these formalizations do not completely coincide.

A second aspect of Semantic Syntax besides transformationality that appears to be essential to the theory is cyclicity. In our view, the fact that the Semantic Syntax structures consist of recursive S-trees that form cyclic domains for (most of) the transformations is not an accidental property of the formalization, but an integral part of the theory. As such, it is reflected both in the generator and in the parser. In the generator, the cycles are processed bottom-up and in the parser, they are processed top-down. In section 6.3.2, we have noted a somewhat problematic consequence of cyclic rules creating new S-nodes, but we have also shown that this
problem can be solved. Further study is needed to determine if we have achieved the optimal solution.

What does not seem to be essential to the theory is the make-up of the Surface Structure, contrary to that of the Semantic Analysis. Obviously, word order is essential to surface structure but the exact constituency structure and depth of node layers seem less fixed. We have shown that a structuralist surface structure like the one AMAZON delivers differs in a number of respects from the Semantic Syntax surface structure, but that these differences do not appear to have any theoretical or practical consequences. We can use a different surface structure like AMAZON's as the input for our parser and still end up with a set of rules that reflect many of the core Semantic Syntax rules as defined by Seuren. In our view, further study of the Semantic Syntax surface structure is needed to determine which aspects of it are well motivated and which are more or less coincidental. For example, as we have already argued (see section 4.4.7), the generative postcyclic rule of Mince which flattens out the surface structure is essentially a cosmetic rule and not very well embedded in the grammar. Now it seems safe to say that no mirror rule is needed in a Semantic Syntax parser. This reinforces the impression that Mince is a superficial rule that perhaps can be dispensed with. Something similar has been claimed with respect to the generative rule Affix Handling, which does not need a counterpart in the parser (see section 6.3.2). Note, however, that for a sound study of the status of the Semantic Syntax surface structure, a better understanding is needed of the morphophonological component and its interface with syntax.

We have demonstrated in this chapter the basic principles of how to implement a Semantic Syntax parser for the basic structures of Dutch. We have shown that the generation of the basic structures can be inverted, although it is not a matter of simply inverting each of the relevant generative rules. Only in some cases have we genuinely mirrored a single generator rule into a single parser rule. For example, the parser rule Participle removes the participle affix (see section 6.3.2), whereas the generator rule Participle adds the participle affix (see sections 4.3.1 and 5.5.3). Both rules apply cyclically and in a highly similar way. In most cases, however, the parallels are smaller. To deal with argument structure, for example, the parser needs more operations than the generator. The generator rule Argument-Functions (see sections 4.2.4 and 5.5.1) is matched by two separate parser rules, one to deal with the subject function and one to deal with the object functions. Additionally, the parser needs three technical rules to retrieve the argument frame from the lexicon and two filter rules (see section 6.3.3).

The question arises to what extent this basic parser can be extrapolated to cover also the more complex phenomena that Seuren (1996) describes. Although we must be cautious with our claims in this respect, it seems fair to say that there is no reason to expect serious problems. The results obtained so far are promising enough to justify the assumption that the same strategy of inverting the transformational process will also hold for the more complex phenomena. Obviously, these complex phenomena may need more intricate rules to handle them correctly, but in principle they will
need similar solutions as already developed within other transformational parsers for Dutch, like the CASUS system (see Coppen 1995). CASUS was developed at the University of Nijmegen, as a post-module to AMAZON (see also footnotes 5, p.15, and 83, p.187). Although its output structures differ greatly from Semantic Analyses, its transformational approach to various phenomena of Dutch syntax is likely to be usable in a Semantic Syntax parser as well. In fact, we have already used part of its approach in the way we analyze argument structure (see section 6.3.3).

What could possibly also be transferred from CASUS to a future full Semantic Syntax parser is the way of putting to good use the features delivered by AMAZON. As I have explained in section 6.3.1, we have been erasing all the featural information from AMAZON’s output structure as a first step towards the SA structure. The reason for doing this was that we did not want our first implementation of Semantic Syntax rules to be guided too much by the availability of extraneous features. However, for various rules within our basic parser we have indicated that there might be ways of improving them by using AMAZON’s feature output. For example, we have argued that the way we analyze finite verbs by looking up the verb forms in a word form lexicon might be improved upon by making use of these features, thus perhaps even rendering the word form lexicon unnecessary (see section 6.3.2). Similarly, many other (existing and future) rules may be implemented more efficiently by drawing on the detailed information that AMAZON delivers as features to every node.

Making use of AMAZON’s featural information may also be of help in avoiding ambiguities. An example of this is given in section 6.3.4, where we have argued that taking subject-verb agreement and wh-status into account might be of help in preventing the introduction of ambiguities by the rule of Fronting. However, ambiguity is a much more general problem than just this. In fact, it is known to be the largest problem faced by parsers (see section 6.1). Ambiguities at various levels in the derivation easily lead to an exponential increase of the number of analyses. Again, we expect a Semantic Syntax parser to be able to use similar strategies as applied within CASUS to avoid this danger of combinatorial explosion. The main strategy applied by CASUS as well as by AMAZON is that of underspecification. Further research should investigate to what extent underspecification can be used within Semantic Syntax parsing as well, but we anticipate no fundamental problems in this respect.

All in all, we may conclude from this thesis that Semantic Syntax can indeed be implemented as a generator or parser in a technolinguistic way, derived from a clear formalization that is well-motivated by linguistic principles. The technolinguistic exercise offers more insight in the distinction between formalization and theory, and the former is improved in some non-trivial respects. It should be remembered, though, that our study did not aim at a technological instrument. Further study is necessary to develop our generator and parser into technological tools that can be plugged into natural language processing systems.
References


Appendix: GRAMGEN

GRAMGEN is an interpreter for Extended Affix Grammars (EAGs), developed by Peter-Arno Coppen at the department of Language and Speech at the University of Nijmegen. It is written in SNOBOL, a string manipulation language, and it runs on DOS/Windows, Apple Macintosh and UNIX platforms. The input for GRAMGEN is an EAG; the output is a generated sentence, in the form of a bracketed structure (which may also be represented as a tree).

Extended Affix Grammars

EAGs belong to the family of two-level grammars; see Meijer (1986) for a formal account, or Aarts and Van den Heuvel (1985, p.313-318) who describe the formalism from a linguistic point of view. EAGs are a direct offspring of two-level Van Wijngaarden grammars, which were originally developed for the formal definition of programming languages: see also Cleaveland and Uzgalis (1977). An EAG consists of two layers, with two context-free grammars laid on top of each other, so to speak. The bottom layer is the actual grammar (also called the hypergrammar) and the top layer consists of the affix level (the metagrammar). The interaction of both layers gives the formalism context-sensitive power. This corresponds to a type 1 grammar in the Chomsky-hierarchy; see Levelt (1974) or Cleaveland and Uzgalis (1977) for a clear exposé on this hierarchy.

The hypergrammar consists of hyperrules: rewrite rules with a left-hand and a right-hand side, separated by a colon. A rewrite rule ends with a full stop. The left-hand side specifies the element to be rewritten by the right-hand side. The right-hand side may consist of zero, one or more members separated by commas. These members may be either terminals or non-terminals. A terminal is a literal enclosed in double quotes; a non-terminal is an element that is to be rewritten and thus is itself the left-hand side of some hyperrule (possibly the same rule in which it is a right-hand side member). The right-hand side may also contain several alternative rewritings, separated by semicolons. An example of a rewrite grammar in EAG notation is:

abc:  a, b, c.
a:    "A", a; .
b:    "B", b; .
c:    "C", c; .

This grammar defines a string of an arbitrary number of As, followed by an arbitrary number of Bs, followed by an arbitrary number of Cs (all possibly zero), for example:
The hypergrammar is combined with a second level in the following manner. A (hyper)non-terminal may be followed by a display between round brackets: a sequence of one or more affix expressions, separated by commas. An affix expression is an affix terminal or an affix non-terminal (or a string of affix terminals and/or non-terminals concatenated by the operator +). An affix terminal is, again, a literal enclosed in double quotes. The domain of the affix non-terminal is defined in the metagrammar. The metarules are distinguished from the hyperrules by the symbol used to separate left-hand from right-hand side: in the metagrammar this symbol is a double colon, instead of a single colon. Furthermore, the symbol to combine members is the concatenation operator +, instead of a comma; alternatives are separated by a semicolon. The consistent substitution constraint demands that each occurrence of an affix non-terminal in a rule represents the same value. The example grammar may be extended to a two-level EAG in the following manner:

\[
\begin{align*}
abc: & \quad a(n), b(n), c(n). \\
a("I"+n): & \quad "A", a(n). \\
a(zero): & \quad . \\
b("I"+n): & \quad "B", b(n). \\
b(zero): & \quad . \\
c("I"+n): & \quad "C", c(n). \\
c(zero): & \quad . \\
zero:: & \quad "." . \\
n:: & \quad zero; "I"+n.
\end{align*}
\]

This grammar, too, defines a string of As, Bs and Cs, but now the number of As equals the number of Bs and the number of Cs. In other words, this grammar defines $a^nb^nc^d$, a notorious example of a language that cannot be described by a context-free (type 2) grammar but can be described by a context-sensitive (type 1) grammar; see also Cleaveland and Uzgališ (1977, p.17). An example tree structure would be:
Non-EAG elements in GRAMGEN

GRAMGEN is basically an EAG-interpreter, but there are some differences from standard EAG notation. First of all, affix expressions are between angled brackets instead of parentheses (parentheses are used for delimiting constituents in the output structure), but of course this is only a minor notational difference.

Second, GRAMGEN needs a unique starting symbol with which to begin the expansion process. This must be defined in a “rule” with no right-hand side, only stating the element that is to be rewritten first (the top node, in tree terminology):

```
abc
```

Furthermore, to express all relevant generalizations in GENIUS, we accommodated GRAMGEN with some extra mechanisms that do not belong to standard EAG either. We introduced a special notation for auxiliary elements that we do not want to appear in the output. A zero attached to a non-terminal ensures that it does not appear in the resulting structure itself, only its daughters will. If we applied this in the hyperrules of the example grammar to the non-terminals a, b and c:

```plaintext
abc: a<n>0, b<n>0, c<n>0.
```

the resulting structure would only contain the top node and terminals, e.g.:

```
 abc
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
</tbody>
</table>
```

We also introduced a distribution operator * for the purpose of distributing an affix across nodes. An affix preceded by a * on more than one sister node is instantiated only on one of the sister nodes. Thus, the following two hyperrules are equivalent:
abc<n>: a<+n>, b<+n>, c<+n>.
abc<n>: a<n>, b, c;
    a, b<n>, c;
    a, b, c<n>.

Another extra mechanism we added is that of a metahierarchy. A metarule may not just define the domain of an affix non-terminal, it may also impose a hierarchical order for the possible rewritings of the non-terminal. This is best explained by means of another example grammar:

phone.
phone:     prefix, number<!figures>.
prefix:    number<![THREE]figures>, dash.
number<multi>: fig, number<figures>.
number"TWO": fig, fig.
fig:       "1"; "2"; "3"; "4"; "5";
    "6"; "7"; "8"; "9"; "0".
dash:      "-".
figures::  ("SEVEN") << "SIX" << "FIVE" << "FOUR" <<
            [THREE] "THREE" << "TWO".
multi::    "SEVEN"; "SIX"; "FIVE"; "FOUR"; "THREE".

This grammar captures the orthography of telephone numbers with a prefix of three figures followed by a number of six or seven figures, e.g. 024-612862, in a binary branching structure (note that this simple example grammar does not express additional constraints such as the first figure of the prefix having to be a zero):

```
+---+---+
| prefix         | number<SIX> |
+---------------+-------------|
| number<THREE> | dash        |
| fig           | number<FIVE>|
+---------------+-------------|
| fig           | number<TWO>|
    |         -   |
| 0 fig         | 6 fig       |
    | 1 fig       |
    | number<THREE>|
      | 2 fig       |
      | number<TWO>|
        | 8 fig fig   |
          | 6 2         |
```

The metarule for the affix non-terminal figures contains the hierarchy. By means of double angled brackets (<<) rather than a semicolon, it expresses a fixed order for the alternative affix terminals: figures must be rewritten as SEVEN first (optionally), then as SIX, then as FIVE and so on. Optional alternatives, such as SEVEN in
the example grammar, are put between parentheses; these parentheses may also be nested when used with more than one alternative, or they may be combined with an asterisk to indicate that an alternative may be chosen more than once.

The hierarchy is first called in the rewrite rule for `phone`, in the affix expression with `number`. The exclamation mark in front of the affix non-terminal indicates that at this point a new hierarchy is initiated: whenever a subordinate constituent contains the same non-terminal, this is substituted according to the order expressed in the hierarchy, starting at the top.

The hyperrule for `prefix` calls a second, independent instantiation of the hierarchy. Again, there is an exclamation mark to force a new substitution process to begin, but this time not at the top of the hierarchy but at the point indicated between square brackets. This so-called entry point is marked as such in the metarule for `figures`. Note that it would not be possible to refer to an alternative in the hierarchy directly rather than to a marked entry point, because the same alternative may occur more than once in a hierarchy.